Desarrollo de Aplicaciones Web

Gleiston Guerrero Ulloa

12 de septiembre de 2025

Índice general

Capítulo 1

Frameworks para Desarrollo Web

Este capítulo presenta un panorama comparativo y analítico de los frameworks y bibliotecas contemporáneos más empleados en el desarrollo web. Antes de abordar cada tecnología, se establecen criterios comunes de análisis: modelo de programación y arquitectura, ecosistema y herramientas, rendimiento y escalabilidad, seguridad, pruebas e integración continua, casos de uso típicos y fortalezas. Con ello se garantiza una lectura coherente y una base objetiva para seleccionar la tecnología adecuada según los requisitos del proyecto, las competencias del equipo y las restricciones no funcionales.

Preparación del Entorno con Node.js y Gestores

Node.js es un entorno de ejecución de JavaScript creado en 2009 por Ryan Dahl con el objetivo de ampliar las capacidades de este lenguaje más allá del navegador. Basado en el motor V8 de Google Chrome, permite ejecutar JavaScript en el lado del servidor y se caracteriza por un modelo de entrada y salida no bloqueante y orientado a eventos, lo que lo hace altamente eficiente para aplicaciones con alta concurrencia. Desde sus primeras versiones, Node.js ha transformado el ecosistema del desarrollo web al unificar el lenguaje entre cliente y servidor, y se ha consolidado como una pieza fundamental en proyectos modernos que requieren escalabilidad, como servidores web, sistemas de mensajería, microservicios y plataformas en tiempo real.

Instalación de Node.js LTS

La relevancia de Node.js en el desarrollo actual no solo radica en su uso directo como servidor mediante frameworks como **Express**, sino también en su papel

como soporte para el *tooling* de front-end. Frameworks como **Angular**, bibliotecas como **React**, y opciones más ligeras como **Vue.js** dependen de Node.js para sus herramientas de construcción, empaquetado, pruebas y despliegue. En este sentido, aunque el código de la aplicación se ejecute finalmente en el navegador, es indispensable disponer de Node.js en el entorno de desarrollo para compilar los proyectos, gestionar dependencias y ejecutar servidores locales con recarga en caliente. Por esta razón, se recomienda siempre instalar la versión **LTS (Long-Term Support)**, que asegura estabilidad y actualizaciones de seguridad durante varios años. El uso de versiones inestables puede ocasionar incompatibilidades con librerías críticas y fallos inesperados en entornos productivos.

Windows

En Windows puede instalarse con el instalador oficial (`.msi`) desde `https://nodejs.org`, siguiendo los siguientes pasos:

1. Ir a https://nodejs.org
2. Descargar el instalador .msi (versión LTS recomendada).
3. Ejecutar el instalador → siguiente, aceptar licencia, dejar las opciones por defecto.
4. Verificar instalación en PowerShell o CMD:

```
node -v
npm -v
```

5. *Consejo:* tras instalar/actualizar Node LTS, reinstale dependencias del proyecto para evitar inconsistencias del binario nativo.

Una alternativa más flexible es utilizar `nvm-windows`, que permite instalar y cambiar entre distintas versiones:

```
nvm install lts
nvm use lts
node -v
npm -v
```

Linux/macOS

En sistemas basados en Unix, como Linux y macOS, el manejo de entornos de desarrollo requiere especial atención a la compatibilidad entre las diferentes versiones de Node.js y sus dependencias. Por ello, una práctica ampliamente recomendada es el uso de gestores de versiones como *Node Version Manager* (NVM),

que permite instalar, alternar y actualizar versiones de manera flexible, reduciendo conflictos con proyectos existentes. Este enfoque resulta particularmente relevante en entornos de desarrollo colaborativo donde distintos equipos trabajan con versiones específicas de Node.js, garantizando homogeneidad en la configuración del entorno [1].

Además, NVM facilita la instalación de versiones LTS (*Long Term Support*), que ofrecen mayor estabilidad y soporte extendido, características clave en entornos productivos donde la fiabilidad es prioritaria.

```
curl -fsSL https://raw.githubusercontent.com/nvm-sh/nvm/master/install.sh | bash
nvm install --lts
nvm use --lts
```

Después de la Instalación...

Preparar correctamente el entorno con Node.js y un gestor de paquetes confiable constituye un requisito indispensable para garantizar estabilidad, productividad y escalabilidad en el desarrollo web moderno.

```
#Después de instalar nodejs
node -v       # Verificar versión instalada
npm -v        # Verificar npm (gestor por defecto que acompaña a Node)
```

Gestores de Paquetes

Junto con Node.js se distribuye **npm (Node Package Manager)**, el gestor de paquetes oficial y más utilizado en el ecosistema. npm permite instalar, actualizar y administrar librerías externas que forman parte de un proyecto, evitando la gestión manual de dependencias. A lo largo del tiempo han surgido alternativas que responden a necesidades específicas de los equipos de desarrollo: **Yarn**, que introdujo mejoras de rendimiento y control de versiones en sus inicios; **pnpm**, que destaca por su eficiencia en el uso de espacio y por su modelo de instalación basado en un almacén global compartido; y **Bun**, un gestor más reciente integrado en un *runtime* alternativo que prioriza la velocidad de ejecución y la simplicidad. Todos ellos cumplen la misma función esencial: garantizar que las dependencias de un proyecto se instalen de manera consistente y reproducible. La elección entre uno u otro depende de factores como el tamaño del equipo, el tipo de proyecto y las necesidades de integración con sistemas de integración continua.

Por lo tanto, resumiendo, los gestores disponibles son: **npm**, incluido por defecto; **pnpm**, que optimiza espacio y velocidad; **yarn**, con *workspaces* y plug'n'play; y **bun**, un runtime alternativo con gestor propio.

Se recomienda usar un solo gestor por proyecto y mantener bajo control el archivo de bloqueo (`package-lock.json`, `pnpm-lock.yaml`, etc.).

Importancia estratégica del entorno

La preparación del entorno de desarrollo web no es un paso trivial, sino un factor determinante en la calidad y sostenibilidad de los proyectos. La correcta elección de la versión de Node.js y del gestor de paquetes garantiza consistencia en la instalación de dependencias, estabilidad frente a actualizaciones y una base sólida para la integración continua. Una configuración inadecuada puede derivar en errores difíciles de depurar, incompatibilidades entre librerías críticas y pérdida de productividad del equipo. Por ello, estandarizar el entorno constituye una práctica esencial para asegurar que el software evolucione de manera controlada y confiable en todas las fases de su ciclo de vida.

Node.js y los Frameworks de Desarrollo Web

La instalación de Node.js es el paso inicial indispensable antes de emprender el desarrollo con frameworks modernos como Angular, React, Vue.js o Express. Tras instalarlo, es esencial verificar la ejecución de los comandos `node -v` y `npm -v`, los cuales confirman tanto la versión activa del motor de ejecución como la del gestor de paquetes. En equipos distribuidos, usar herramientas como **nvm (Node Version Manager)** o su equivalente **nvm-windows** asegura uniformidad de entorno entre distintos desarrolladores, evitando incompatibilidades que afectan la compilación, el rendimiento y la estabilidad del entorno de desarrollo. Ante un cambio de versión de Node.js, se recomienda reinstalar las dependencias del proyecto mediante `npm ci`, `pnpm install --frozen-lockfile` o `yarn install --frozen-lockfile`, para reconstruir correctamente los módulos nativos y garantizar compatibilidad total.

Una vez instalado y verificado el entorno, es habitual ejecutar un proyecto de prueba en cada framework para asegurar que la construcción, el empaquetado y el servidor local funcionan correctamente. Node.js por sí solo no gestiona directamente el consumo de servicios REST, pero es el soporte fundamental que permite a frameworks como Angular y Express compilar, ejecutar y desplegar aplicaciones que luego interactúan con APIs back-end, como las RESTful implementadas en Spring Boot[1], usando clientes HTTP tipados. Esta interacción con APIs RESTful, basada en peticiones HTTP y manejo de JSON, se desarrolla con mayor precisión en la sección de Angular [3], [4], [5].

[1]Spring Boot es un marco de trabajo de código abierto basado en Spring que simplifica la creación de aplicaciones Java al proporcionar configuración automática, un servidor embebido (como Tomcat o Jetty) y un modelo de empaquetado listo para producción. Su objetivo principal es reducir la complejidad del desarrollo y despliegue de aplicaciones empresariales, permitiendo a los desarrolladores enfocarse en la lógica de negocio en lugar de en la configuración extensa del entorno [2].

Por otra parte, frameworks como React y Vue.js suelen utilizar herramientas como Vite para un servidor de desarrollo con recarga en caliente y empaquetado optimizado; mientras que Express permite levantar rápidamente un servidor backend para verificar que Node.js está funcionando correctamente del lado del servidor. Estas comprobaciones iniciales no solo confirman que el entorno está listo, sino que proporcionan una base estable y reproducible sobre la cual se construyen aplicaciones más complejas.

Dependencias para Evitar Inconsistencias de Binarios Nativos

En el ecosistema de Node.js, las dependencias incluyen tanto módulos JavaScript como complementos con binarios nativos dependientes del sistema operativo y la versión de Node.js. Por ello, al cambiar de versión de Node o trabajar en entornos distintos, pueden surgir errores al compilar o ejecutar el proyecto. La reinstalación de dependencias —como vía `npm ci`, `pnpm install --frozen-lockfile` o `yarn install --frozen-lockfile`— resulta esencial para garantizar coherencia, estabilidad y reproducibilidad, ayudando a reconstruir módulos nativos correctamente sin ambigüedad en la versión usada [6], [7].

El uso de herramientas como `nvm use --lts` en Linux/macOS o su equivalente para Windows (`nvm use lts` con nvm-windows) permite alinear a todos los desarrolladores a una misma versión LTS de Node.js, evitando incompatibilidades en la ABI (Interfaz Binaria de Aplicación). Así se propicia un entorno homogéneo y predecible, especialmente importante cuando los módulos contienen componentes compilados de manera nativa para distintas plataformas [8], [9].

Si los módulos ya instalados están corruptos o inconsistentes —por interrupciones durante la instalación o migración entre sistemas—, puede eliminarse manualmente la carpeta `node_modules` (ej. con `rm -rf node_modules` en Unix o `Remove-Item -Recurse -Force node_modules` en Windows) y reinstalar desde los archivos de bloqueo (`package-lock.json`, `pnpm-lock.yaml`, etc.), reconstruyendo un entorno limpio y alineado con las versiones estrictas definidas [10], [11].

En entornos de integración continua, el comando `npm ci` es especialmente valioso porque reinstala las dependencias con alta fidelidad al archivo de bloqueo, sin recalcular versiones. En contraste, `npm install` sirve más para instalaciones iniciales o actualizaciones en rangos permitidos, reflejando dichos cambios en el lockfile. Asimismo, comandos como `npm rebuild` (recompilación de módulos nativos) y `npm cache verify / npm cache clean` contribuyen a preservar la integridad del entorno frente a posibles corrupciones o fallos en dependencias [6], [10].

Gestores alternativos como **pnpm** y **Yarn** también ofrecen versiones equivalentes que priorizan reproducibilidad y eficiencia. Por ejemplo: - `pnpm install --frozen-lockfile` deja intacto el lockfile y evita diffs no deseados, - `pnpm rebuild` recompila

módulos nativos y `pnpm store prune` limpia versiones antiguas, - De forma similar, `yarn install --frozen-lockfile`, `yarn rebuild` y `yarn cache clean` soportan flujos idénticos en proyectos colaborativos o entornos monorepo [11], [12].

Finalmente, una alternativa emergente como **Bun** incluye comandos propios como `bun install --frozen-lockfile` y `bun rebuild`. Aunque más reciente, su enfoque en reproducibilidad, rendimiento y manejo consistente de módulos nativos lo hace prometedor para equipos que priorizan velocidad y consistencia en sus entornos de desarrollo [13], [14].

En conjunto, estas prácticas permiten reducir al mínimo las discrepancias entre entornos locales, de pruebas y de producción. Reinstalar dependencias de forma sistemática ante cambios de versión, o cuando se detectan inconsistencias, constituye una medida preventiva que asegura un desarrollo más confiable y profesional en proyectos basados en Node.js y frameworks asociados.

Instalación y Reinstalación de Node.js

La correcta instalación de Node.js constituye la base de trabajo para frameworks de desarrollo web como Angular, React, Vue.js o Express. Existen diversas formas de instalarlo según el sistema operativo y la flexibilidad que se requiera. En Windows, el método más directo es el instalador oficial (`.msi`), que incluye también a **npm** como gestor por defecto. Sin embargo, para entornos en los que se necesita alternar entre versiones, la mejor práctica es utilizar gestores de versiones como **nvm-windows** en Windows o **nvm** en Linux/macOS, que permiten instalar múltiples versiones y activar la adecuada según el proyecto en curso. En entornos de servidores o contenedores también es común usar **NodeSource**, repositorios oficiales para distribuciones Linux, o incluso imágenes de **Docker**, cuando se desea aislar la ejecución en contenedores reproducibles.

```
    # Linux/macOS con NVM
    curl -fsSL https://raw.githubusercontent.com/nvm-sh/nvm/master/install.sh | bash
    nvm install --lts
    nvm use --lts

    # Windows con nvm-windows
    nvm install lts
    nvm use lts

    # Instalador oficial (Windows, macOS, Linux)
    # Descargar desde https://nodejs.org
```

Una vez instalado, la verificación debe realizarse comprobando las versiones activas de Node.js y de npm:

```
    node -v    # versión activa de Node.js
    npm -v     # versión activa de npm
```

En caso de inconsistencias al cambiar de versión de Node.js, especialmente en proyectos que usan módulos con binarios nativos, es recomendable reinstalar las dependencias desde cero. El procedimiento típico consiste en eliminar la carpeta `node_modules`, reinstalar dependencias a partir del archivo de bloqueo y recompilar módulos nativos.

```
# Linux/macOS
rm -rf node_modules

# Windows (PowerShell)
Remove-Item -Recurse -Force node_modules

# Reinstalación limpia con npm
npm ci
npm rebuild
```

Además de npm, existen gestores alternativos que ofrecen comandos equivalentes y beneficios adicionales:

- **pnpm**:

```
pnpm install --frozen-lockfile
pnpm rebuild
pnpm store prune   # limpia versiones no usadas
```

- **Yarn**:

```
yarn install --frozen-lockfile
yarn rebuild
yarn cache clean
```

- **Bun** (runtime alternativo):

```
bun install --frozen-lockfile
bun rebuild
```

La elección entre estos gestores dependerá de las necesidades del proyecto: **npm** es universal y estable; **pnpm** optimiza espacio y velocidad; **Yarn** es útil en monorepos; y **Bun** representa una alternativa emergente centrada en rendimiento.

En todos los casos, mantener un único gestor por proyecto y conservar actualizado el archivo de bloqueo (`package-lock.json`, `pnpm-lock.yaml`, `yarn.lock`, `bun.lockb`) son prácticas fundamentales para garantizar coherencia y reproducibilidad en equipos de desarrollo y entornos de integración continua. Esta disciplina reduce discrepancias entre entornos locales, de pruebas y de producción, asegurando que Angular, React, Vue.js y Express funcionen de manera estable sobre Node.js.

Angular

Angular se ha consolidado como una plataforma integral para construir interfaces modernas, escalables y mantenibles, gracias a su arquitectura basada en componentes, al tipado de TypeScript y a un *tooling* estandarizado que reduce la variabilidad entre proyectos. En el contexto de aplicaciones empresariales que consumen servicios RESTful, Angular ofrece un cliente HTTP tipado, enrutamiento declarativo, formularios reactivos y utilidades para internacionalización, pruebas e integración continua, lo que facilita un flujo de trabajo coherente desde el prototipo hasta la puesta en producción [15], [16], [17].

Descripción y Propósito

Angular es una plataforma de desarrollo *front-end* mantenida por Google que proporciona un conjunto integrado de herramientas para construir aplicaciones web de una sola página (SPA) con TypeScript, HTML y CSS. Se orienta a equipos y bases de código grandes, con un fuerte énfasis en tipado estático, patrones consistentes y *tooling* corporativo [15], [18].

Configuración del Ambiente de Desarrollo

Angular requiere Node.js LTS y un gestor de paquetes compatible. Para instrucciones detalladas, véase la sección 1. Por lo tanto, los pasos que son necesarios ejecutar para obtener una aplicación web con Angular y algún Back-End son los siguientes:

```
# 1) Instalar Node LTS (vía NVM recomendado) y verificar:
#(revisar sección 1 de este capítulo)
#Comprobar versiones
node -v
npm -v

# 2) Instalar Angular CLI global con npm (sencillo para empezar):
npm install -g @angular/cli
ng --version

# 3) Crear proyecto indicando gestor (pnpm como ejemplo) y opciones básicas:
ng new tienda-app --package-manager=npm --routing --style=scss
cd tienda-app

# 4) Servir en desarrollo:
ng serve -o

# 5) Construcción de producción:
ng build --configuration production
```

Resultado: aplicación Angular lista para desarrollo local y con artefactos en `dist/` para desplegar.

Ejemplo de Uso y Tooling

En la práctica del desarrollo con frameworks modernos como Angular, no basta únicamente con instalar las dependencias principales, sino que se requiere un ecosistema de herramientas que complementen el flujo de trabajo. Entre ellas se incluyen sistemas de construcción, gestores de paquetes, linters y formateadores de código, así como servidores de desarrollo en caliente. Estas herramientas, en conjunto, permiten acelerar la iteración, garantizar la calidad del código y facilitar la integración continua en proyectos colaborativos [19].

Instalar Node JS LTS

La instalación y reinstalación de Node JS se detalla en la sección 1 y en la sección 1. Antes de instalar la última versión de Node JS (versión LTS) se debe verificar la versión instala y activa de node y el gestor de paquetes escogido para esta tarea.

```
node -v       # Verificar versión instalada
npm -v        # Verificar npm (gestor por defecto que acompaña a Node)
```

Instalar Angular CLI

[20]: La *Angular Command Line Interface* (CLI) constituye una herramienta esencial para la creación y gestión de proyectos en Angular, ya que automatiza tareas comunes como la generación de componentes, servicios, módulos y pruebas. Además, ofrece comandos optimizados para la construcción del proyecto y la preparación para entornos de despliegue, lo que reduce significativamente el riesgo de errores manuales.

Al estar disponible a través de diferentes gestores de paquetes (`npm`, `pnpm`, `yarn`, `bun`), brinda flexibilidad para adaptarse a distintas preferencias de los equipos de desarrollo y a las estrategias de manejo de dependencias en proyectos de gran escala.

```
# con npm
npm install -g @angular/cli

# alternativas soportadas
pnpm install -g @angular/cli
yarn global add @angular/cli
bun install -g @angular/cli
```

Crear el proyecto

La creación de un nuevo proyecto en Angular no se limita a la ejecución de un simple comando, sino que implica la configuración inicial de toda la estructura de carpetas, archivos de configuración y dependencias necesarias para el ciclo de vida de la aplicación. Con la opción `--routing`, se habilita de inmediato el módulo de enrutamiento, facilitando la navegación entre diferentes vistas desde el inicio del proyecto. Asimismo, el uso de `--style=scss` permite configurar *Sass* como preprocesador de estilos, proporcionando una sintaxis más avanzada y organizada en comparación con CSS plano [21].

Esta combinación acelera la adopción de buenas prácticas en equipos de desarrollo, ya que fomenta el uso de estilos modulares y la separación clara de responsabilidades entre lógica y presentación. Además, una vez generado el proyecto, comandos como `ng serve -o` permiten iniciar un servidor de desarrollo local con recarga automática, optimizando el flujo iterativo de construcción y prueba en entornos ágiles.

```
ng new tienda-app --routing --style=scss
cd tienda-app
ng serve -o
```

Extensiones útiles de editor

Entre las extensiones más comunes por su utilidad se pueden mencionar a Angular Language Service, ESLint y Prettier [22], [23], [24]. Estas herramientas se han consolidado como complementos indispensables en entornos de desarrollo profesional, ya que proporcionan asistencia inteligente en la edición de plantillas, aseguran la calidad del código mediante reglas de estilo y garantizan un formateo homogéneo en los proyectos colaborativos [25].

El **Angular Language Service** proporciona autocompletado, información de tipos y validación en tiempo real dentro de plantillas HTML de Angular. Esto reduce errores comunes de sintaxis y mejora la productividad al ofrecer sugerencias contextuales basadas en los metadatos del proyecto [22].

La extensión **ESLint** cumple un rol fundamental en el aseguramiento de la calidad del código, ya que permite aplicar reglas de estilo y verificar el cumplimiento de convenciones en el equipo. Su integración con Angular garantiza que las aplicaciones se construyan bajo criterios de consistencia, mantenibilidad y reducción de defectos técnicos [26].

Por su parte, **Prettier** complementa a ESLint automatizando el formateo del código de acuerdo con reglas predefinidas, evitando discusiones en torno al estilo y permitiendo que los equipos se concentren en la lógica de la aplicación. Esta integración asegura que el código se mantenga legible y homogéneo en repositorios

compartidos, lo que resulta crucial en proyectos colaborativos y de larga duración [27].

Habilitar HttpClient y formularios

La habilitación del cliente HTTP y de los formularios constituye un paso esencial en cualquier aplicación Angular moderna. El cliente HTTP (`HttpClient`) permite establecer comunicación tipada y segura con servicios externos a través de peticiones RESTful, mientras que el sistema de formularios ofrece dos enfoques complementarios: formularios reactivos, orientados a un control explícito de los flujos de datos, y formularios basados en plantillas, adecuados para escenarios más declarativos.

Ambos enfoques aprovechan el sistema de tipado de TypeScript y la detección de cambios de Angular, garantizando consistencia entre la lógica de negocio y la interfaz de usuario [28]. En el contexto de aplicaciones Standalone (Angular 17+), la configuración del cliente HTTP y del enrutador se centraliza en el archivo `main.ts`, lo que simplifica la inicialización del proyecto y concentra la definición de proveedores esenciales en un único punto de arranque.

```
/* src/main.ts */
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { provideHttpClient, withFetch, withXsrfConfiguration } from
    '@angular/common/http';
import { provideAnimations } from '@angular/platform-browser/animations';

import { AppComponent } from './app/app.component';
import { routes } from './app/app.routes';

bootstrapApplication(AppComponent, {
    providers: [
        provideRouter(routes),
        provideHttpClient(
            withFetch(),
            withXsrfConfiguration({ headerName: 'X-XSRF-TOKEN', cookieName:
    'XSRF-TOKEN' })
        ),
        provideAnimations()
    ]
}).catch(err => console.error(err));
```

Listing 1.1: Arranque Standalone en `main.ts`.

Arquitectura y Buenas Prácticas

Angular adopta una arquitectura basada en componentes, inyección de dependencias (DI) y un enrutador propio; favorece patrones reactivos mediante RxJS. Sus elementos nucleares incluyen componentes (vista y lógica), servicios (lógica de negocio compartida), directivas/*pipes* (composición y transformación) y *routing*. El compilador transforma plantillas en código optimizado durante la construcción (*build*) [29], [30].

Arquitectura de Angular

La Figura 1.1 ilustra de manera esquemática el flujo de comunicación entre una aplicación Angular en el cliente y un backend desarrollado en Spring Boot.

La representación refleja la interacción jerárquica y modular de las capas, destacando cómo Angular organiza la interfaz y la lógica de presentación, mientras que Spring Boot gestiona los procesos de negocio y la persistencia de datos. Este enfoque arquitectónico no solo favorece la separación de responsabilidades, sino que también garantiza la escalabilidad y el mantenimiento del sistema [29].

En la parte superior se ubica el navegador, donde se ejecuta la aplicación Angular. Dentro de este contexto, los **componentes** constituyen la unidad fundamental de la interfaz de usuario, encargados de enlazar las plantillas con los datos dinámicos y gestionar la interacción del usuario. Estos componentes delegan la lógica de negocio a los **servicios**, que centralizan la comunicación con el backend utilizando el módulo `HttpClient` y la librería `RxJS` para el manejo reactivo de flujos de datos.

Entre los servicios y la capa de transporte HTTP se sitúan los **interceptores**, que cumplen un rol esencial en la gestión de peticiones y respuestas. Por medio de interceptores es posible añadir cabeceras de autenticación, registrar solicitudes, reintentar conexiones en caso de fallos o configurar dinámicamente las URL base de los endpoints. Este diseño aporta un punto de control transversal en la aplicación, alineándose con buenas prácticas de seguridad y trazabilidad [31].

De manera complementaria, el **Router de Angular** gestiona la navegación entre vistas, implementando rutas, guardias de acceso y resolvers para la precarga de datos antes de la activación de un componente. Esto garantiza una experiencia de usuario fluida, en la que las transiciones entre páginas se realizan sin recargar la aplicación completa.

En la parte inferior, el diagrama muestra la conexión con la **API REST** implementada en Spring Boot. Esta capa expone los recursos de la aplicación en formato JSON a través de HTTP, los cuales son consumidos por los servicios de Angular. Una vez recibida la petición, la capa de servicio en Spring procesa la lógica de negocio y delega las operaciones de persistencia al **repositorio**, típicamente implementado con JPA. Finalmente, los datos se almacenan o recuperan desde la base de datos, cerrando el ciclo de comunicación entre cliente y servidor [3], [4].

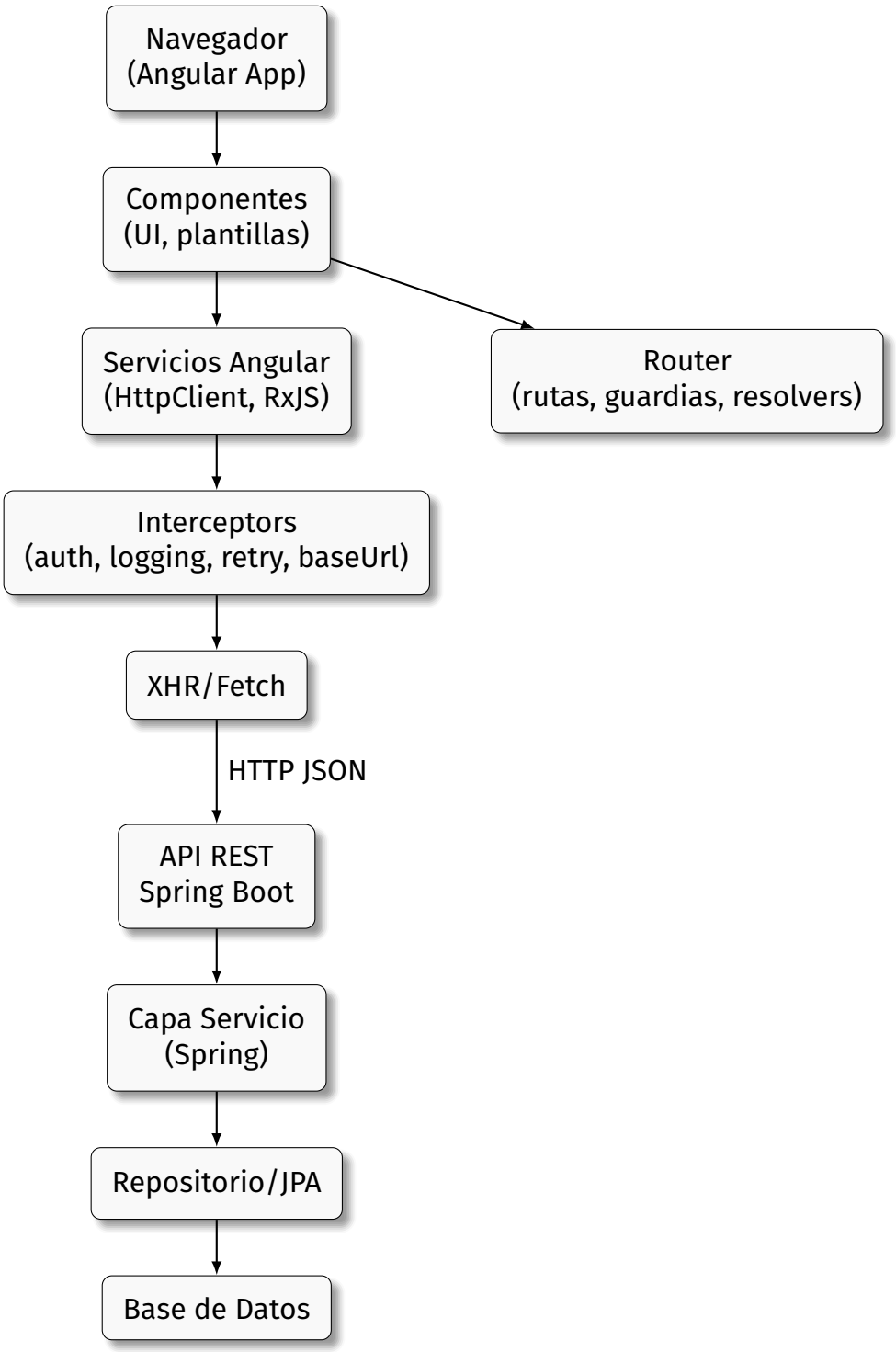

Figura 1.1: Arquitectura de consumo de API REST con Angular (front-end) y Spring Boot (back-end).

Desarrollo de un Proyecto Angular

Un proyecto en Angular consiste en un conjunto organizado de carpetas y archivos base que conforman el denominado *workspace*. Este incluye la aplicación principal, la configuración de compilación, los entornos de pruebas, los archivos de tipado como `tsconfig`, el manifiesto de dependencias en `package.json`, así como los recursos estáticos requeridos para su ejecución. Esta estructura estandarizada ofrece a los equipos de desarrollo una base sólida y repetible, lo que reduce la curva de aprendizaje inicial y evita errores frecuentes derivados de configuraciones manuales [32].

Entre sus características destacan el soporte opcional para enrutamiento, la preconfiguración de estilos mediante opciones como SCSS, una organización centrada en componentes y la presencia de scripts automatizados para construcción y pruebas. La disponibilidad de estas características permite que los desarrolladores, incluso sin experiencia previa, puedan producir aplicaciones escalables, mantenibles y alineadas con buenas prácticas de ingeniería de software.

Las funcionalidades principales de un proyecto Angular incluyen el punto de entrada `main.ts`, el componente inicial `AppComponent`, la configuración de entornos definida en `environment`, y la arquitectura basada en inyección de dependencias (DI) y módulos o configuraciones *standalone*. Este mecanismo de inyección de dependencias facilita el desacoplamiento entre módulos y promueve la reutilización de código, lo cual es esencial en proyectos de gran escala.

En los proyectos generados con Angular CLI, la estructura inicial cumple un rol esencial al estandarizar la organización de archivos y carpetas, lo que facilita la curva de aprendizaje y garantiza coherencia entre equipos de desarrollo.

A partir de Angular 17, la arquitectura recomendada es la basada en *Standalone APIs*, por lo que aparecen archivos como `app.config.ts` y `app.routes.ts`, que reemplazan la necesidad del tradicional `AppModule` [33]. No obstante, cuando se crea un proyecto con la opción `--standalone=false`, todavía se incluye un `app.module.ts`, manteniendo compatibilidad con versiones anteriores (Angular 16 o previas). Este cambio refleja una evolución en el framework hacia la simplificación y la modularidad explícita, reduciendo dependencias implícitas y mejorando la eficiencia en tiempo de compilación [32].

Por lo tanto, comprender estas diferencias entre las versiones es fundamental en contextos académicos y profesionales, ya que permite analizar la transición de Angular desde una arquitectura centrada en módulos hacia un paradigma más moderno y ligero, basado en configuración directa en `main.ts` y en el uso de `bootstrapApplication` para el arranque de la aplicación [29].

Nota: La presencia de `environments/` y `app.config.ts` puede variar según la versión de la CLI y las opciones del generador; son archivos habituales pero opcionales en la plantilla Standalone moderna.

```
tienda-app/
├── node_modules/
├── src/
│   ├── app/
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.component.ts
│   │   ├── app.config.ts
│   │   └── app.routes.ts
│   ├── assets/
│   ├── environments/
│   │   ├── environment.ts
│   │   └── environment.prod.ts
│   ├── favicon.ico
│   └── main.ts
├── angular.json
├── package.json
├── tsconfig.app.json
├── tsconfig.json
├── tsconfig.spec.json
└── README.md
```

Figura 1.2: Estructura inicial de un proyecto Angular generado con la CLI (`ng new`).

Servidor de desarrollo (*dev server*) con recarga en caliente

El servidor de desarrollo en Angular constituye una herramienta fundamental para la fase inicial de programación, ya que permite servir la aplicación de manera local en memoria y recompilar automáticamente cuando se detectan cambios en el código fuente. Esta capacidad hace posible que el desarrollador visualice en tiempo real cada modificación realizada en las plantillas, los estilos o la lógica de los componentes, sin necesidad de reiniciar el servidor manualmente. El efecto inmediato de los cambios agiliza notablemente el proceso de construcción de interfaces y facilita la detección temprana de errores.

Una de sus características más destacadas es el *Hot Module Replacement* (HMR), conocido en español como “reemplazo en caliente de módulos”. Este mecanismo evita la recarga completa de la aplicación, conservando el estado interno mientras únicamente se actualizan las secciones modificadas [34]. Gracias a ello, es posible mantener la continuidad de los flujos de trabajo y acelerar la validación de fun-

cionalidades. Este comportamiento se complementa con el uso de *source maps*, que permiten depurar el código escrito en TypeScript observando directamente las líneas originales, en lugar del JavaScript transpilado, lo que reduce significativamente la complejidad durante las tareas de depuración.

En términos funcionales, el servidor de desarrollo facilita una navegación inmediata dentro de la aplicación para validar tanto la interfaz de usuario como los flujos de interacción. Además, ofrece integración con la configuración de un *proxy*, lo que resulta útil para redirigir solicitudes hacia un back-end y evitar problemas relacionados con CORS durante la etapa de desarrollo. Esto implica que incluso un programador principiante puede simular de forma sencilla la comunicación con un servidor sin necesidad de configurar mecanismos complejos.

Su uso práctico se ejemplifica en la edición de componentes o vistas en el editor de código y la observación de resultados instantáneos en el navegador. En muchos proyectos, se emplea un archivo de configuración como `proxy.conf.json` para redirigir automáticamente todas las peticiones realizadas a rutas específicas, como `/api`, hacia el servidor de back-end. De esta manera, el desarrollador únicamente modifica la interfaz de usuario en el entorno local y obtiene retroalimentación inmediata, lo que optimiza la productividad y asegura un flujo de trabajo más eficiente.

Construcción de producción (*build* optimizado)

La construcción de producción en Angular corresponde al proceso mediante el cual la aplicación se transforma en un conjunto de artefactos estáticos preparados para ser desplegados en un entorno real. Estos artefactos se generan en la carpeta `dist/` y contienen versiones optimizadas del código fuente que garantizan un rendimiento adecuado y una carga eficiente en el navegador. Este procedimiento constituye una fase esencial, ya que marca la transición entre el desarrollo local y la disponibilidad pública de la aplicación.

Durante la construcción, se aplican técnicas de optimización como la minificación de archivos y la eliminación de código muerto mediante *tree-shaking*. Asimismo, el sistema divide el código en fragmentos asociados a rutas o módulos específicos, lo que permite cargar dinámicamente solo las partes necesarias de la aplicación. Estas prácticas no solo reducen el tamaño de los archivos descargados, sino que también mejoran la experiencia del usuario final al disminuir los tiempos de carga y optimizar el uso de los recursos del sistema.

Otro aspecto relevante de la construcción es la integración de variables de `environment`, que permite diferenciar configuraciones entre los entornos de desarrollo y producción. De esta forma, el mismo proyecto puede adaptarse a distintas necesidades sin modificar manualmente los archivos de código. El sistema incluye, además, la verificación de presupuestos de *bundle*, lo que asegura que los tamaños de los paquetes generados no sobrepasen límites establecidos previamente,

contribuyendo a mantener la eficiencia y escalabilidad de la aplicación.

El resultado de este proceso es un conjunto de archivos HTML, JavaScript y CSS listos para desplegarse en un servidor estático, una red de distribución de contenido (CDN) o en plataformas especializadas de hosting, como Nginx o Firebase Hosting. Una vez publicados, estos artefactos garantizan que los usuarios accedan a una versión estable, segura y optimizada de la aplicación, preparada para responder de manera eficiente a las demandas de un entorno productivo.

Modelo `models/producto.model.ts`

El archivo `core/models/producto.model.ts` cuyo contenido se muestra en el listado de código 1.2, cumple la función de definir la estructura de datos asociada a los productos dentro de la aplicación Angular. Se trata de un modelo tipado en TypeScript, cuyo objetivo es estandarizar cómo se representan los objetos en memoria y cómo se intercambian a través de los distintos componentes, servicios y módulos de la aplicación. Al establecer un contrato explícito para las propiedades que debe poseer un producto, se reduce la ambigüedad en el código y se incrementa la robustez en la comunicación entre cliente y servidor.

```
1  //src/app/core/models/producto.model.ts
2
3  export interface Producto {
4      id: number;
5      nombre: string;
6      precio: number;
7      descripcion?: string; // opcional
8      stock?: number;
9  }
```

Listing 1.2: Modelo de dominio `Producto`.

Contrato JSON con el back-end. La interfaz `Producto` define el contrato de datos que se intercambia con la API. Para garantizar compatibilidad, los nombres de las propiedades deben corresponderse con las claves del JSON emitido por Spring Boot (p. ej., `id`, `nombre`, `precio`). Si el back-end expone nombres distintos (por ejemplo, `price` en lugar de `precio`), se recomienda adaptar el DTO en el servidor o mapear la respuesta en el cliente antes de exponerla al resto de la aplicación, a fin de mantener un modelo de dominio consistente en el front-end.

Este modelo ofrece ventajas claras en términos de mantenibilidad, ya que cualquier cambio en la definición de un producto —por ejemplo, agregar un nuevo campo como `categoria` o `stock`— puede centralizarse en este archivo, propagándose de manera automática a todas las partes del sistema que lo utilizan. Además, permite aprovechar al máximo el sistema de tipos de TypeScript, proporcionando autocompletado, detección temprana de errores y mayor claridad semántica en el código

fuente.

Además, este modelo actúa como contrato de datos entre el cliente Angular y la API REST [3], [4] de Spring Boot, ya que refleja la misma estructura de atributos utilizada en el formato JSON del servidor. De esta manera, se asegura que la comunicación entre ambos extremos sea consistente, evitando errores por discrepancias en los tipos de datos o nombres de campos [16], [35].

Dentro de la arquitectura de la aplicación, el modelo `Producto` actúa como la base de la comunicación entre el `ProductosService` y los componentes de interfaz, ya que define la forma exacta de los datos que se reciben desde la API y que posteriormente se presentan en la UI. En consecuencia, constituye un punto de convergencia entre la lógica de negocio y la representación visual, garantizando coherencia y seguridad de tipos en todo el flujo de datos de la aplicación.

En aplicaciones de mediana o gran escala, este enfoque modular resulta fundamental porque evita la duplicación de definiciones en distintos puntos del sistema, facilita el trabajo colaborativo y asegura la compatibilidad entre las distintas capas de la aplicación [5], [36], [37].

Componente `productos/lista`

Antes de presentar la plantilla principal de la aplicación, conviene ilustrar un ejemplo de componente protegido. El `DashboardComponent` constituye la vista de acceso restringido, normalmente asociada a un área interna de la aplicación. Su propósito es mostrar información sensible únicamente a usuarios autenticados y con permisos adecuados.

El listado de código 1.3 muestra la definición mínima de este componente en modo Standalone.

```
/* src/app/dashboard/dashboard.component.ts */
import { Component } from '@angular/core';

@Component({
    selector: 'app-dashboard',
    standalone: true,
    template: `<h1>Dashboard</h1><p>Zona protegida.</p>`
})
export class DashboardComponent {}
```

Listing 1.3: `DashboardComponent` Standalone.

De manera complementaria al panel de control, el `LoginComponent` (ver Listado de código 1.4) proporciona la interfaz inicial para la autenticación de usuarios. Se trata de un componente Standalone que expone un formulario de acceso o, en su versión básica, un mensaje indicando la necesidad de iniciar sesión. Su implementación sencilla facilita la integración posterior con servicios de autenticación más avanzados.

```ts
/* src/app/login/login.component.ts */
import { Component } from '@angular/core';

@Component({
    selector: 'app-login',
    standalone: true,
    template: `<h1>Login</h1><p>Autentíquese para continuar.</p>`
})
export class LoginComponent {}
```

Listing 1.4: `LoginComponent` Standalone.

Para manejar los intentos de acceso no autorizado, se utiliza el `ForbiddenComponent` cuya lógica de negocios se muestra en el Listado de código 1.5. Este componente Standalone actúa como página de error dedicada a usuarios que, si bien están autenticados, no poseen los privilegios necesarios para acceder a determinadas rutas. De esta forma, se refuerza la separación entre autenticación y autorización dentro de la aplicación.

```ts
/* src/app/forbidden/forbidden.component.ts */
import { Component } from '@angular/core';

@Component({
    selector: 'app-forbidden',
    standalone: true,
    template: `<h1>403</h1><p>Acceso denegado.</p>`
})
export class ForbiddenComponent {}
```

Listing 1.5: `ForbiddenComponent` Standalone.

El `AppComponent` constituye el componente raíz de la aplicación Angular. En el enfoque Standalone, este componente importa explícitamente `RouterOutlet` para habilitar la navegación entre vistas, lo que garantiza que el enrutamiento definido en `app.routes.ts` se renderice dinámicamente en la interfaz. El Listado de código 1.6 muestra su definición.

```ts
/* src/app/app.component.ts */
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';

@Component({
    selector: 'app-root',
    standalone: true,
    imports: [RouterOutlet],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.scss']
})
export class AppComponent {}
```

Listing 1.6: Componente raíz Standalone `AppComponent`.

La plantilla asociada al componente raíz es sumamente concisa, ya que delega la responsabilidad de la navegación en el enrutador de Angular. Mediante la directiva `<router-outlet>`, se insertan dinámicamente los componentes correspondientes a las rutas activas. Este patrón refuerza la modularidad y la separación de responsabilidades en la arquitectura de la aplicación.El listado de código 1.7 muestra la inclusión de la directiva `<router-outlet>` como parte de la plantilla raíz, cuya función es actuar como contenedor dinámico de los componentes enrutados.

```
<!-- src/app/app.component.html -->
<router-outlet></router-outlet>
```

Listing 1.7: Plantilla raíz con `RouterOutlet`.

El componente `productos/lista` (ver Listados 1.8 y 1.9) constituye la pieza de interfaz encargada de proyectar en pantalla los datos consumidos desde la API REST. Este componente se suscribe al flujo de productos expuesto por el servicio, y mediante el uso de `AsyncPipe` y `trackBy`, renderiza eficientemente los registros devueltos por el servidor. De esta forma, el front-end Angular refleja en tiempo real la información proveniente del back-end, manteniendo la separación entre la lógica de datos (servicio) y la presentación (componente) [38], [39].

Vinculación UI ↔ API. Los componentes de presentación consumen el `Observable<Producto[]>` expuesto por `ProductosService` y proyectan sus datos en la plantilla mediante `AsyncPipe`. En la clase del componente (listado 1.8), la propiedad `productos$` recibe el flujo emitido por `listar()`, mientras que en la plantilla (listado 1.9) se recorre la colección con `*ngFor`, aplicando `trackBy` para minimizar el trabajo de detección de cambios. Esta separación de responsabilidades mantiene el componente *delgado* y delega en el servicio la interacción con la API.

```
/* src/app/productos/lista/lista.component.ts */
import { Component, ChangeDetectionStrategy } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Observable } from 'rxjs';
import { ProductosService } from '../../core/productos.service';
import { Producto } from '../../core/models/producto.model';

@Component({
    selector: 'app-productos-lista',
    standalone: true,
    imports: [CommonModule],
    templateUrl: './lista.component.html',
    styleUrls: ['./lista.component.scss'],
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class ListaComponent {
    productos$: Observable<Producto[]> = this.productosSvc.listar();
    constructor(private productosSvc: ProductosService) {}
    trackById = (_: number, p: Producto) => p.id;
    seleccionar(p: Producto): void { console.log('Seleccionado:', p); }
```

```
        editar(p: Producto): void { console.log('Editar:', p); }
}
```

Listing 1.8: `ListaComponent` Standalone con `CommonModule`.

Este enfoque admite flujos de trabajo declarativos mediante enlaces de datos, que incluyen interpolación, binding de propiedades y binding de eventos, permitiendo sincronizar la vista con el modelo sin manipulación manual del DOM. La detección de cambios (change detection) opera automáticamente, actualizando el DOM cuando el estado del componente varía [40]. Además, Angular permite renderizar listas dinámicamente mediante directivas estructurales como `*ngFor` e incorpora condicionales en la vista con `*ngIf`, facilitando la creación de estructura dinámica en la interfaz [41].

En su funcionamiento, el componente inyecta un servicio que proporciona un observable con una lista de productos (por ejemplo, `Observable<Producto[]>`), y la plantilla recorre dicha colección utilizando `*ngFor`, donde puede aplicarse `trackBy` para optimizar el rendimiento. Cada elemento del listado puede incluir botones o acciones (como "seleccionar" o "editar") enlazados mediante eventos, lo que permite comunicar interacciones hacia componentes superiores. Esta arquitectura separa claramente la responsabilidad de presentación del acceso a datos—la lógica reside en el servicio, mientras el componente solo maneja la visualización y la interacción.

El listado 1.9 muestra la plantilla del componente con `AsyncPipe` y `trackBy`, que renderiza eficientemente la colección proveniente del servicio. La lógica TypeScript correspondiente se presenta en el listado 1.8, donde se habilita `ChangeDetectionStrategy.OnPush` y se expone el flujo `productos$`.

```
<!-- src/app/productos/lista/lista.component.html -->

<section class="productos">
    <h2>Productos</h2>

    <ng-container *ngIf="productos$ | async as productos; else cargando">
        <div *ngIf="productos.length; else sinDatos">
            <ul>
                <li *ngFor="let p of productos; trackBy: trackById">
                    <strong>{{ p.nombre }}</strong>
                    - {{ p.precio | currency:'USD' }}
                    <button type="button"
    (click)="seleccionar(p)">Seleccionar</button>
                    <button type="button" (click)="editar(p)">Editar</button>
                </li>
            </ul>
        </div>
    </ng-container>

    <ng-template #cargando>
        <p>Cargando...</p>
```

```
    </ng-template>

    <ng-template #sinDatos>
        <p>No hay productos para mostrar.</p>
    </ng-template>
</section>
```

Listing 1.9: Plantilla del componente `productos/lista` con `AsyncPipe` y `trackBy`.

Arranque Standalone en `main.ts`

En esta configuración, `provideHttpClient` actúa como punto de integración con la API REST de Spring Boot, mientras que `provideRouter` establece la navegación de la SPA. El front-end permanece desacoplado del back-end: basta con ajustar `environment.apiBaseUrl` (o un `proxy` de desarrollo) para redirigir el tráfico HTTP al servidor correspondiente sin tocar el código de los componentes.

El punto de entrada Standalone (ver Listado 1.1) utiliza la función `bootstrapApplication` y registra tanto el enrutador como el cliente HTTP mediante `provideRouter` y `provideHttpClient`. Esta configuración habilita la integración directa del front-end Angular con el back-end RESTful en Spring Boot, asegurando que todas las rutas declaradas en `app.routes.ts` se comuniquen con la API de manera tipada y consistente.

A partir de Angular 17, las APIs *Standalone* se han convertido en la forma recomendada para crear aplicaciones sin necesidad de utilizar `NgModule`. En esta guía se adopta plenamente el enfoque *Standalone*: las rutas se definen en `app.routes.ts` y se registran en `main.ts` mediante `provideRouter`. El arranque de la aplicación se realiza con `bootstrapApplication`, y cada componente enrutado se declara con la opción `standalone: true`. Este cambio simplifica la estructura del proyecto, mejora la legibilidad del código y refleja las mejores prácticas actuales en el ecosistema Angular.

Consumo de API RESTful desde Angular

Un cliente Angular interactúa con una API RESTful mediante peticiones HTTP tipadas que intercambian JSON. Este esquema promueve la separación de responsabilidades: el cliente gestiona estado de vista, interacción y composición de UI; el servidor define recursos, reglas de negocio y persistencia. La asincronía se representa con *Observables* de RxJS, lo que habilita composición, cancelación y manejo uniforme de éxito/errores.

En entornos profesionales, se recomienda parametrizar la URL base del back-end vía `environments`, emplear un `proxy` en desarrollo para evitar CORS, y centralizar en servicios el acceso a datos y el tratamiento de errores.

Servicio `core/productos`

El servicio `core/productos` constituye la capa de acceso a datos y lógica compartida para la gestión de productos, siendo el componente responsable de conectar la aplicación con el back-end. Su propósito es aislar los detalles de comunicación HTTP del resto de la aplicación, de manera que los componentes se mantengan enfocados en la presentación y en la interacción con el usuario. Este enfoque favorece la separación de responsabilidades y permite una mayor mantenibilidad del código en aplicaciones de mediana y gran escala [36].

Fundamentos del consumo REST en Angular. En Angular, el consumo de una API RESTful se realiza típicamente mediante `HttpClient`, que emite *Observables* de RxJS para representar respuestas asincrónicas. El intercambio de datos se efectúa en formato JSON y, por convención, cada recurso se identifica mediante una ruta estable (p. ej., `/api/productos`) y operaciones CRUD accesibles vía HTTP (`GET`, `POST`, `PUT`, `DELETE`). Este patrón, alineado con la arquitectura cliente-servidor, favorece la independencia entre el front-end Angular y el back-end (API RESTful implementada en Spring Boot), simplifica pruebas y despliegues, y habilita optimizaciones como *caching* y reintentos controlados en el cliente [3], [4], [35].

Una de sus principales características es que se declara como inyectable mediante el mecanismo de inyección de dependencias (DI) propio de Angular. Esto significa que el servicio puede ser reutilizado en múltiples componentes sin necesidad de instanciarlo manualmente. Además, define métodos tipados para las operaciones más comunes sobre productos —como `get`, `post`, `put` y `delete`— garantizando la coherencia de tipos gracias al uso de TypeScript.

Las funcionalidades de este servicio incluyen la encapsulación de las URLs de la API, la definición de parámetros de las solicitudes y el mapeo de las respuestas obtenidas. También centraliza operaciones avanzadas como reintentos de conexión, almacenamiento en caché y manejo de errores. Para un desarrollador principiante, esto se traduce en una ventaja significativa: no es necesario repetir en cada componente la lógica de conexión al servidor, ya que todo se concentra en un punto común y reutilizable dentro de la aplicación.

En la práctica, el servicio `core/productos` suele implementar métodos como `listar()`, `obtener(id)`, `crear(dto)`, `actualizar(id, dto)` y `eliminar(id)`. Estos métodos son consumidos desde los componentes o resolvers de Angular. Por ejemplo, un componente de lista puede invocar de manera directa a `productosService.listar()` para obtener los datos y mostrarlos en la interfaz, sin necesidad de preocuparse por los detalles técnicos de la comunicación HTTP. Este nivel de abstracción mejora la productividad del equipo y reduce la probabilidad de errores relacionados con la duplicación de lógica de red en diferentes partes del sistema [5].

Transferencia de Representaciones del Estado de los Recursos o *Representational State Transfer* (REST)

REST (*Representational State Transfer*) es un estilo arquitectónico propuesto por Roy Fielding en el año 2000, caracterizado por restricciones como la comunicación cliente-servidor, la ausencia de estado, el uso de una interfaz uniforme y la posibilidad de almacenamiento en caché [3], [4]. Estas reglas definen un marco teórico que ha servido de base para la evolución de la web moderna y la construcción de servicios distribuidos.

La literatura técnica y fuentes aplicadas coinciden en destacar que RESTful representa un mayor grado de conformidad con los principios de REST, garantizando uniformidad y consistencia en el diseño de las interfaces [42], [43]. En la práctica, esta distinción ayuda a evaluar la calidad arquitectónica de un servicio y su capacidad para integrarse en entornos distribuidos, escalables y confiables.

Diferencia entre REST y RESTful

El término **RESTful** hace referencia a aquellas implementaciones de servicios que cumplen estrictamente con todos los principios de REST. En contraste, algunas APIs denominadas únicamente "REST" pueden no adherirse completamente a dichas restricciones, aplicando solo un subconjunto de ellas [44], [45]. De esta forma, mientras REST describe el estilo y sus principios, RESTful designa la fidelidad de una API a dicho estilo.

Presentación del listado 1.10. El listado 1.10, correspondiente al archivo `src/app/core/productos.service.ts`, muestra la implementación del servicio `ProductosService` que orquesta las operaciones CRUD contra la API RESTful implementada en Spring Boot. La URL base del back-end se obtiene desde los archivos de `environments`, lo que permite diferenciar entornos (desarrollo/producción) sin modificar el código de la capa de acceso a datos. Además, cada método retorna un `Observable<T>` tipado, de modo que los componentes puedan suscribirse (o usar `AsyncPipe`) y reaccionar a nuevas emisiones, errores o finalización de la secuencia [16], [46].

```
1  // src/app/core/productos.service.ts
2  import { Injectable } from '@angular/core';
3  import { HttpClient } from '@angular/common/http';
4  import { Observable } from 'rxjs';
5  import { Producto } from './models/producto.model';
6
7  @Injectable({
8      providedIn: 'root'
9  })
10 export class ProductosService {
```

```
    private baseUrl = '/api/productos'; // URL base de la API

    constructor(private http: HttpClient) {}

    listar(): Observable<Producto[]> {
        return this.http.get<Producto[]>(this.baseUrl);
    }

    obtener(id: number): Observable<Producto> {
        return this.http.get<Producto>(`${this.baseUrl}/${id}`);
    }

    crear(producto: Omit<Producto, 'id'>): Observable<Producto> {
        return this.http.post<Producto>(this.baseUrl, producto);
    }

    actualizar(id: number, producto: Producto): Observable<Producto> {
        return this.http.put<Producto>(`${this.baseUrl}/${id}`, producto);
    }

    eliminar(id: number): Observable<void> {
        return this.http.delete<void>(`${this.baseUrl}/${id}`);
    }
}
```

Listing 1.10: Servicio core/productos en Angular para gestionar operaciones CRUD.

Además, gracias a la anotación `@Injectable({ providedIn: 'root' })`, el servicio está disponible de forma global en la aplicación, evitando configuraciones adicionales y facilitando su reutilización en cualquier componente que lo requiera.

Guardia de rutas `auth/auth`

En Angular, un guardia de rutas constituye un mecanismo fundamental para controlar el acceso a secciones específicas de una aplicación. Se implementa mediante interfaces como `CanActivate`, `CanDeactivate` o `CanLoad`, que permiten determinar si una ruta debe activarse, cargarse o abandonarse en función de condiciones predefinidas [47]. Este enfoque resulta especialmente útil en aplicaciones que requieren protección de áreas privadas o sensibles, como paneles administrativos o flujos restringidos a determinados perfiles de usuario.

La característica central de los guardias de rutas es su integración directa con el enrutador de Angular. Gracias a esta integración, es posible consultar de manera directa el estado de autenticación del usuario o validar sus roles antes de renderizar un componente o cargar un módulo de forma diferida (*lazy loading*). Cuando las condiciones no se cumplen, el guardia redirige la navegación hacia rutas alternativas, como la página de inicio de sesión, evitando así accesos no autorizados [48].

Su funcionalidad no se limita únicamente a la autenticación. Los guardias también pueden utilizarse para verificar la validez de datos en formularios, la disponibilidad de servicios o incluso la configuración de permisos a nivel de organización. De este modo, los desarrolladores obtienen una herramienta versátil para aplicar reglas de negocio relacionadas con la navegación y la seguridad de la aplicación [49].

Un caso práctico habitual consiste en verificar el estado de sesión mediante un servicio de autenticación. Si la función `isLoggedIn()` (véase el Listado de código 1.11) devuelve `true`, el guardia concede acceso a la ruta solicitada. En entornos empresariales, este mecanismo puede ampliarse para implementar políticas más estrictas, como la validación de roles específicos antes de autorizar el acceso a determinados recursos, lo que permite aplicar un modelo de control de acceso basado en roles (RBAC).

```
// src/app/auth/auth.guard.ts
import { Injectable } from '@angular/core';
import { CanActivate, Router, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';
import { AuthService } from './auth.service';

@Injectable({
    providedIn: 'root'
})
export class AuthGuard implements CanActivate {

    constructor(private authService: AuthService, private router: Router) {}

    canActivate():
    | boolean
    | UrlTree
    | Observable<boolean | UrlTree>
    | Promise<boolean | UrlTree> {

        // Verificar si el usuario está autenticado
        if (this.authService.isLoggedIn()) {
            // Opcional: validar roles
            if (this.authService.hasRole('ADMIN')) {
                return true;
            } else {
                // Redirigir a página de "acceso denegado"
                return this.router.createUrlTree(['/forbidden']);
            }
        }

        // Si no está autenticado, redirigir al login
        return this.router.createUrlTree(['/login']);
    }
}
```

Listing 1.11: Guardia de rutas `auth/auth.guard.ts` en Angular para gestionar accesos y roles.

El listado de código 1.12 muestra un servicio de autenticación básico que proporciona el estado de sesión y la verificación de roles, sobre el cual se apoya el guardia de rutas.

```
// src/app/auth/auth.service.ts
import { Injectable } from '@angular/core';

@Injectable({
    providedIn: 'root'
})
export class AuthService {

    // Simulación de estado de sesión
    private user = {
        loggedIn: true,
        role: 'ADMIN'
    };

    isLoggedIn(): boolean {
        return this.user.loggedIn;
    }

    hasRole(role: string): boolean {
        return this.user.role === role;
    }
}
```

Listing 1.12: Servicio auth/auth.service.ts en Angular para gestionar las sesiones.

En entornos empresariales, donde es común tener múltiples perfiles de usuario (administradores, editores, visitantes), los guardias de rutas resultan imprescindibles para asegurar la segmentación de funciones y el cumplimiento de políticas de seguridad. El listado de código 1.12 muestra un servicio de autenticación básico que gestiona el estado de sesión y la verificación de roles, sobre el cual se apoya el guardia de rutas.

La combinación de sencillez en la implementación y solidez en los resultados convierte a este mecanismo en un pilar esencial dentro de la arquitectura de seguridad de Angular. Su integración con el enrutador proporciona un control granular sobre la navegación, permitiendo no solo validar la autenticación del usuario, sino también aplicar reglas de negocio relacionadas con la autorización y la gestión de permisos. Gracias a esta capacidad de interceptar y condicionar el flujo de enrutamiento, los guardias garantizan que las aplicaciones web desarrolladas con Angular mantengan altos niveles de seguridad, coherencia y confiabilidad, incluso en escenarios complejos de uso real [47], [48], [49].

Enrutamiento Standalone con `app.routes.ts`

En aplicaciones Standalone (Angular 17/18), el enrutamiento se declara en un archivo de rutas exportadas y se registra mediante `provideRouter` en `main.ts`, sin necesidad de un *NgModule*. El siguiente listado define rutas públicas (`/login`, `/forbidden`) y una ruta protegida (`/dashboard`) con el guard `AuthGuard`.

El Listado de código 1.13, correspondiente al archivo `src/app/app.routes.ts`, define la configuración de rutas en Angular bajo el enfoque *Standalone*. Dichas rutas no solo delimitan vistas públicas (como el inicio de sesión o la página de acceso denegado), sino también zonas protegidas que consumen datos expuestos por el back-end REST desarrollado en Spring Boot. Esta separación garantiza que la navegación del cliente esté controlada mediante guardias de autenticación, mientras que los recursos del servidor se mantienen accesibles únicamente bajo las condiciones establecidas. De esta manera, el enrutador se convierte en un elemento clave de integración entre la interfaz de usuario y la API RESTful.

```
/* src/app/app.routes.ts */
import { Routes } from '@angular/router';
import { AuthGuard } from './auth/auth.guard';
import { LoginComponent } from './login/login.component';
import { ForbiddenComponent } from './forbidden/forbidden.component';
import { DashboardComponent } from './dashboard/dashboard.component';

export const routes: Routes = [
    { path: 'login', component: LoginComponent },
    { path: 'forbidden', component: ForbiddenComponent },
    {
        path: 'dashboard',
        component: DashboardComponent,
        canActivate: [AuthGuard] // Ruta protegida
    },
    { path: '**', redirectTo: 'login' }
];
```

Listing 1.13: Rutas Standalone en `app.routes.ts`.

Estructura Final del Proyecto Angular

La Figura 1.3 presenta la estructura final de un proyecto en Angular construido bajo el enfoque *Standalone* (Angular 17+). Este esquema refleja no solo los archivos generados automáticamente por la CLI mediante el comando `ng new`, sino también los directorios y ficheros creados de forma manual durante el desarrollo de la aplicación, tales como servicios, modelos, componentes adicionales y guardias de rutas.

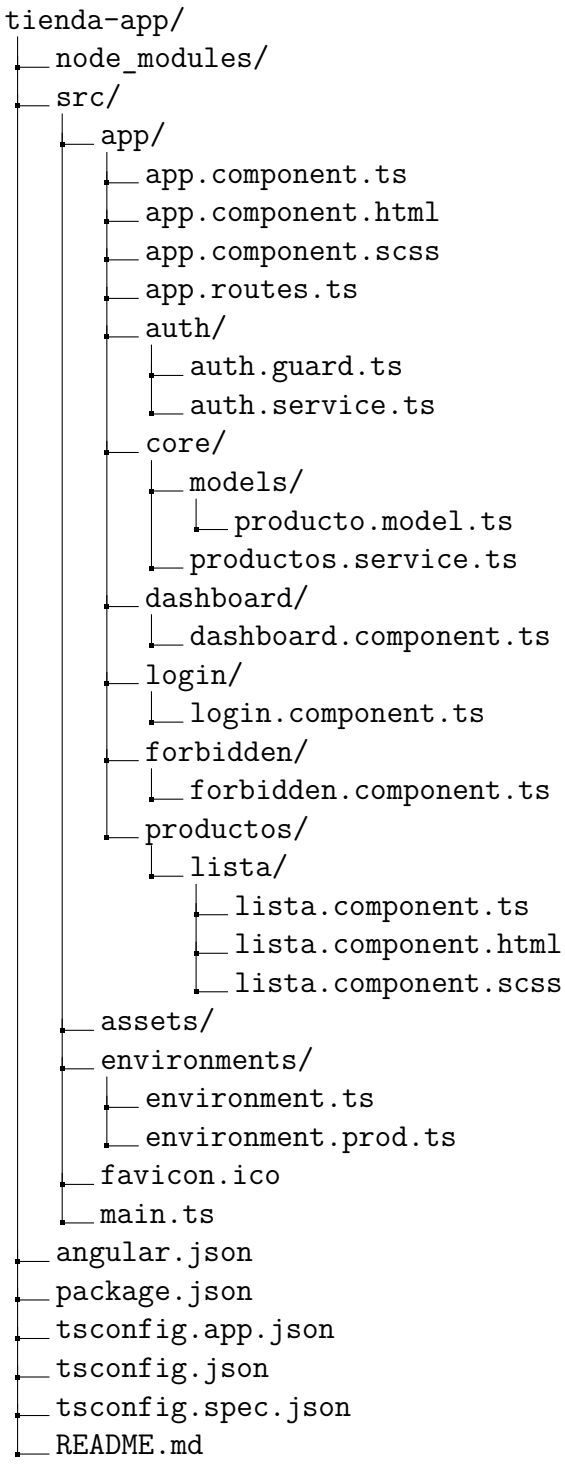

Figura 1.3: Estructura final del proyecto Angular (Standalone) con todos los archivos creados y referenciados en el capítulo.

Nota sobre entornos y CORS

En desarrollo, es habitual emplear un archivo `proxy.conf.json` para redirigir las solicitudes realizadas a `/api` hacia `http://localhost:8080`, evitando así problemas de CORS cuando se ejecuta `ng serve`. En producción, la URL base del back-end se gestiona a través de `environment.prod.ts` (`apiBaseUrl`), de modo que la aplicación Angular pueda apuntar a un servidor desplegado sin cambios en el código. Esta estrategia permite alternar entre entornos sin modificar los servicios ni los componentes, reforzando la portabilidad y mantenibilidad del proyecto.

En primer lugar, se observa el directorio `src/app`, que constituye el núcleo de la aplicación. Allí se ubican tanto el componente raíz (`AppComponent`) como los subdirectorios que agrupan funcionalidades específicas: `auth/` para la lógica de autenticación (incluyendo el guardia de rutas y el servicio correspondiente), `core/` para servicios reutilizables y modelos de dominio, y `productos/` que contiene el componente de lista encargado de mostrar los datos obtenidos desde el back-end. De igual forma, los directorios `dashboard/`, `login/` y `forbidden/` agrupan los componentes asociados a cada vista principal de la aplicación, siguiendo una organización modular que facilita la mantenibilidad y la escalabilidad.

Por otra parte, la carpeta `environments/` contiene los archivos de configuración para distintos contextos de ejecución (desarrollo y producción), mientras que `assets/` agrupa los recursos estáticos necesarios para la interfaz (como imágenes o íconos). Los ficheros de configuración a nivel raíz —`angular.json`, `tsconfig.json`, `package.json`, entre otros— permiten orquestar la compilación, gestionar dependencias y definir parámetros de tipado en TypeScript, garantizando la reproducibilidad del proyecto en distintos entornos.

El archivo `main.ts`, junto con `app.routes.ts`, desempeña un rol central en la arquitectura Standalone: el primero constituye el punto de entrada que inicializa la aplicación mediante la función `bootstrapApplication`, mientras que el segundo define el conjunto de rutas de navegación y aplica las reglas de acceso asociadas a los guardias. De esta manera, el proyecto combina la simplicidad estructural de Angular moderno con la robustez necesaria para escenarios empresariales, asegurando buenas prácticas de modularidad, separación de responsabilidades y control de dependencias.

Finalmente, es importante subrayar que esta estructura no solo responde a una necesidad técnica, sino que representa un estándar académico y profesional para el desarrollo de software moderno. La claridad en la organización de los archivos fomenta la colaboración en equipos multidisciplinarios, reduce la curva de aprendizaje para nuevos desarrolladores y facilita las prácticas de ingeniería de software basadas en pruebas, control de versiones y despliegue continuo. Por ello, comprender y aplicar esta organización es un componente esencial en la formación de competencias en desarrollo web avanzado, particularmente en entornos universi-

tarios y en proyectos de investigación orientados a la transferencia tecnológica y la innovación.

React

React es una biblioteca de JavaScript centrada en la construcción de interfaces mediante componentes funcionales y un modelo declarativo de actualización de la vista. Su contribución arquitectónica más influyente es la abstracción del DOM mediante un árbol virtual y un proceso de *reconciliación* que determina de manera eficiente los cambios mínimos a aplicar, simplificando el modelo de programación y, a la vez, mejorando el rendimiento observable en la práctica [50]. En términos de ingeniería de software, React se alinea con el paradigma de programación reactiva, donde los flujos de datos y la propagación de cambios permiten mantener la coherencia entre estado, lógica y representación [51].

Principios y Modelo de Componentes

El modelo de React parte de componentes funcionales que reciben `props`, gestionan `state` de manera local y producen una descripción declarativa de la UI. La composición de componentes resulta natural al estar basada en objetos y funciones de JavaScript, lo que favorece la modularidad y la reutilización a gran escala [50]. Asimismo, la introducción de *Hooks* formaliza un mecanismo para encapsular y reutilizar lógica de estado y efectos secundarios sin recurrir a clases, reforzando la separación de preocupaciones y la testabilidad del código.

Entorno y Gestión de Versiones

Para asegurar reproducibilidad y consistencia entre equipos, se recomienda fijar versiones de Node.js y dependencias con un gestor de versiones (v. gr., NVM) y bloquear resoluciones en el gestor de paquetes (véase la discusión de entorno en la Sección 1). Esta disciplina de entorno es especialmente relevante en ecosistemas con herramientas de construcción modernas, dado que pequeñas divergencias de versión pueden alterar el comportamiento de la cadena de construcción y del *bundler* subyacente [52].

En el caso de React, se sugiere crear un archivo de configuración que deje constancia de la versión de Node.js utilizada y del gestor de paquetes adoptado en el proyecto. Esto puede lograrse, por ejemplo, con un archivo `.nvmrc`, el cual asegura que todos los miembros del equipo utilicen la misma versión al activar el entorno. El listado 1.14 muestra un ejemplo de este archivo.

```
# Versión LTS recomendada para proyectos React
```

```
    lts/*
    # O una versión específica
    v20.11.1

```

Listing 1.14: Archivo .nvmrc para fijar la versión de Node.js en un proyecto React.

De manera complementaria, es recomendable bloquear la resolución de dependencias del proyecto mediante los archivos de bloqueo (`package-lock.json` en npm o `pnpm-lock.yaml` en pnpm). Estos ficheros garantizan que las mismas versiones de librerías se instalen en cada máquina y en cada despliegue, evitando errores difíciles de reproducir. El listado 1.15 ilustra la inicialización de un proyecto React con Vite y la creación automática de dicho archivo de bloqueo.

```
    # Crear un nuevo proyecto con Vite + React
    npm create vite@latest mi-app-react -- --template react
    cd mi-app-react

    # Instalar dependencias (se crea package-lock.json)
    npm install

```

Listing 1.15: Inicialización de un proyecto React con Vite que genera automáticamente el archivo de bloqueo de dependencias.

Finalmente, para reforzar la consistencia del entorno, puede añadirse al proyecto un archivo `.npmrc` (o su equivalente en otros gestores), que especifique configuraciones clave como la política de versiones semánticas o la fuente del registro de paquetes. Esto permite alinear las configuraciones locales con las de la integración continua y el entorno de producción, minimizando sorpresas en el ciclo de vida de la aplicación.

Creación de un Proyecto con Vite

A diferencia de otros *toolchains*, React adopta un enfoque agnóstico a la herramienta de construcción. Una opción liviana y ampliamente usada es Vite, que aprovecha módulos ES y una arquitectura de desarrollo rápido. Antes de ejecutar el listado 1.16, es pertinente señalar que la selección de plantilla y gestor de paquetes debe alinearse con las políticas del proyecto (tipado con TypeScript, estrategias de linting y formato, etc.).

```
    # Crear proyecto con plantilla React + TypeScript
    npm create vite@latest tienda-react -- --template react-ts
    cd tienda-react
    npm install
    npm run dev

```

Listing 1.16: Inicialización de un proyecto React con Vite.

Tras ejecutar el procedimiento del listado 1.16, Vite crea una organización inicial de archivos y carpetas mostrada en la Figura 1.4. Esta estructura separa la configuración de compilación del código fuente y los recursos estáticos.

```
tienda-react/
   index.html
   package.json
   tsconfig.json
   vite.config.ts
   .gitignore
   node_modules/   (dependencias instaladas; carpeta grande, generada)
   src/
   main.tsx        (punto de entrada; monta <App />)
   App.tsx         (componente raíz)
   App.css         (estilos iniciales)
   assets/         (recursos estáticos)
```

Figura 1.4: Estructura inicial de un proyecto React creado con Vite y TypeScript.

La Figura 1.5 resume el flujo de entorno reproducible para React, evitando inconsistencias entre desarrollo, CI/CD y producción mediante la fijación de Node.js con NVM, el uso de un único gestor de paquetes y el bloqueo de dependencias.

La Figura 1.5 representa de manera esquemática el flujo recomendado para configurar un entorno de trabajo reproducible en proyectos React. En ella se observa cómo la selección de la versión de Node.js mediante NVM y la adopción de un único gestor de paquetes convergen en la generación de archivos de bloqueo de dependencias, los cuales son reutilizados tanto en las estaciones de desarrollo como en los entornos de integración continua y despliegue en producción. Este diseño asegura que las dependencias se resuelvan de forma idéntica en todas las fases del ciclo de vida del software, reduciendo la posibilidad de inconsistencias y fallos difíciles de depurar.

Arquitectura Lógica: Estado, Efectos y Reconciliación

El flujo de datos en React es unidireccional: los padres pasan propiedades a los hijos, mientras que el estado local y el contexto se utilizan para gestionar información transversal. El *render* produce un árbol virtual que React compara (*diff*) con

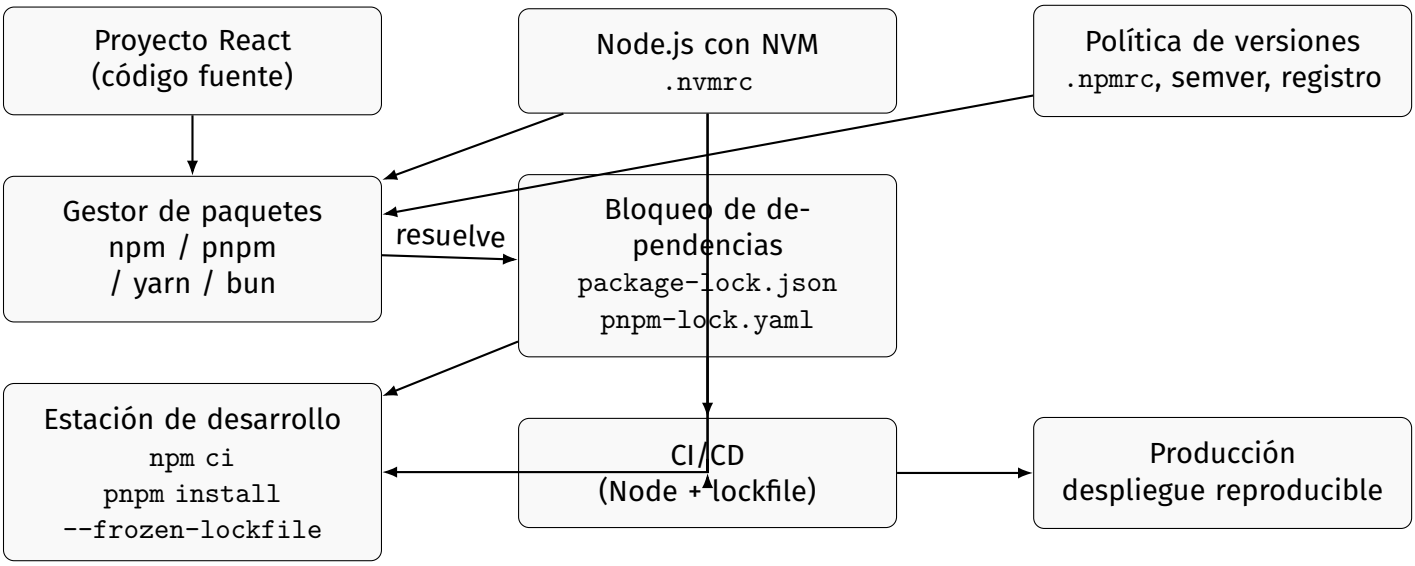

Figura 1.5: Flujo de entorno reproducible para React: fijación de Node.js con NVM, uso de un único gestor de paquetes, bloqueo de dependencias y aplicación consistente en desarrollo, CI/CD y producción.

el árbol previo para aplicar únicamente los cambios necesarios en el DOM real, lo cual reduce trabajo de manipulación y latencias de actualización [50]. El paradigma subyacente de propagación de cambios se corresponde con patrones estudiados en programación reactiva, donde la semántica de flujos y dependencias hace explícitas las fuentes de actualización [51].

Ejemplo de Componente con *Hooks* y Consumo de API

El listado 1.17 muestra un componente funcional que consume una API REST y maneja estado y efectos con `useState` y `useEffect`. Este patrón ilustra la separación entre obtención de datos, estado de carga y representación, así como el tratamiento de abortos de petición para evitar actualizaciones sobre componentes desmontados.

```
import { useEffect, useState } from "react";

type Producto = { id: number; nombre: string; precio: number };

export function ListaProductos() {
    const [data, setData] = useState<Producto[]>([]);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState<Error | null>(null);

```

```
    useEffect(() => {
        const ctrl = new AbortController();
        async function run() {
            try {
                setLoading(true);
                const resp = await fetch("/api/productos", { signal: ctrl.signal });
                if (!resp.ok) throw new Error(`HTTP ${resp.status}`);
                const json = (await resp.json()) as Producto[];
                setData(json);
            } catch (e) {
                if ((e as any).name !== "AbortError") setError(e as Error);
            } finally {
                setLoading(false);
            }
        }
        run();
        return () => ctrl.abort();
    }, []);

    if (loading) return <p>…Cargando</p>;
    if (error)   return <p>Fallo: {error.message}</p>;

    return (
    <ul>
    {data.map(p => (
        <li key={p.id}>
        {p.nombre} - ${"{p.precio.toFixed(2)}"}
        </li>
        ))}
    </ul>
    );
}
```

Listing 1.17: Componente React con Hooks para consumo de API REST.

Gestión de Estado Compartido con Context y *Custom Hooks*

Cuando varias ramas del árbol requieren el mismo estado (tema visual, sesión, carrito), `Context` evita el *prop drilling*. En combinación con *custom hooks* se encapsula la lógica para su reutilización sistemática. El listado 1.18 muestra un patrón minimalista que separa proveedor y consumo de contexto.

```
import { createContext, useContext, useState } from "react";

type Session = { user: string } | null;

const SessionCtx = createContext<Session>(null);
```

```
export function SessionProvider({ children }: { children: React.ReactNode }) {
    const [session, setSession] = useState<Session>(null);
    // ... lógica de login/logout que actualiza 'session'
    return <SessionCtx.Provider value={session}>{children}</SessionCtx.Provider>;
}

export function useSession(): Session {
    return useContext(SessionCtx);
}
```

Listing 1.18: Contexto de sesión y *custom hook* para su consumo.

Pruebas de Componentes

La verificación se apoya comúnmente en *React Testing Library*, que promueve pruebas orientadas al comportamiento del usuario. Antes del listado 1.19, cabe subrayar que este enfoque desaconseja acoplarse a detalles de implementación, privilegiando selectores accesibles y aserciones sobre el resultado observable.

```
import { render, screen } from "@testing-library/react";
import { ListaProductos } from "./ListaProductos";

test("muestra indicador de carga y luego listado", async () => {
    // Se puede simular fetch con MSW o jest.spyOn(global, "fetch") ...
    render(<ListaProductos />);
    expect(screen.getByText(/Cargando/)).toBeInTheDocument();
    // ... esperar al contenido real según la simulación
});
```

Listing 1.19: Prueba de componente centrada en comportamiento observable con React Testing Library.

Consideraciones de Rendimiento

El rendimiento de aplicaciones web modernas está influido por múltiples factores: tamaño del *bundle*, estrategias de carga, caché y el patrón de actualizaciones del árbol de UI. La literatura reciente en ingeniería de software móvil y web ofrece evidencia empírica sobre cómo técnicas de *caching* y *service workers* impactan consumo energético y tiempos de carga, directrices de utilidad también para aplicaciones React orientadas a experiencias progresivas [52]. Estas evidencias refuerzan la necesidad de medir en contexto (perfilado de renders, descomposición de código y límites de contexto/efectos) en lugar de asumir beneficios a priori.

Arquitectura de Alto Nivel

La Figura 1.6 resume un flujo típico en React: componentes funcionales → *hooks* (estado/efectos) → contexto/servicios de acceso a datos → capa de transporte HTTP. El árbol virtual resultante se reconcilia con el DOM para materializar cambios mínimos. Esta abstracción del DOM y el razonamiento por componentes son las claves de su modelo arquitectónico [50], [51].

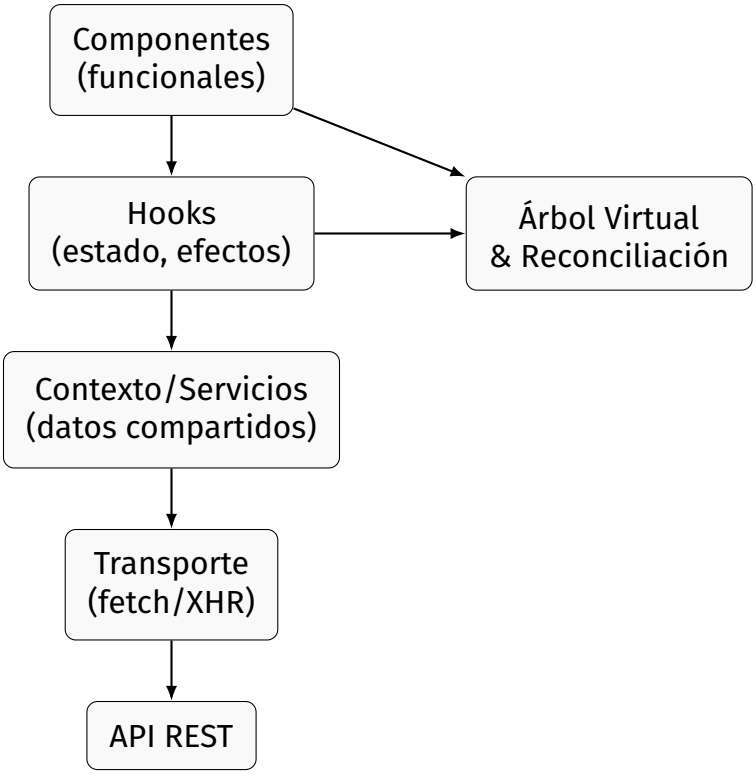

Figura 1.6: Esquema de alto nivel del flujo en React: composición de componentes, gestión de estado/efectos y reconciliación del árbol virtual.

Bibliografía

[1] B. Griggs, *Node Cookbook: Discover solutions, techniques, and best practices for server-side web development with Node.js 14, 4th Edition*. Packt Publishing, 2020, ISBN: 9781838554576. dirección: https://books.google.com.ec/books?id=kW8LEAAAQBAJ

[2] Spring Team. "Spring Boot Reference Documentation." Accedido: 11 de septiembre de 2025. dirección: https://docs.spring.io/spring-boot/docs/current/reference/html/

[3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Accedido: 11 septiembre 2025, Tesis doct., University of California, Irvine, 2000. dirección: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[4] R. Fielding. "Chapter 5: Representational State Transfer (REST)." Accedido: 11 de septiembre de 2025. dirección: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[5] Angular Documentation Team. "Communicating with Backend Services using HTTP," visitado 10 de sep. de 2025. dirección: https://angular.dev/guide/http

[6] M. Anand. "npm i vs npm ci: Understanding the Difference." Accedido: 11 de septiembre de 2025. dirección: https://dev.to/manthanank/npm-i-vs-npm-ci-understanding-the-difference-2pfk

[7] jbandi y RobC. "How to use npm ci instead of npm install for deterministic project setup." Accedido: 11 de septiembre de 2025. dirección: https://stackoverflow.com/questions/53469032/using-npm-ci-instead-of-npm-install-for-deterministic-project-setup

[8] Node.js Foundation. "Node.js Documentation," visitado 9 de sep. de 2025. dirección: https://nodejs.org/en/docs/

[9] NVM Project. "Node Version Manager Documentation," visitado 9 de sep. de 2025. dirección: https://github.com/nvm-sh/nvm

[10] npm, Inc. "npm Documentation," visitado 9 de sep. de 2025. dirección: https://docs.npmjs.com/

[11] pnpm contributors. “pnpm install (CLI) — frozen-lockfile behavior.” Accedido: 11 de septiembre de 2025. dirección: https://pnpm.io/cli/install

[12] Yarn Project. “Yarn Package Manager Documentation,” visitado 9 de sep. de 2025. dirección: https://yarnpkg.com/getting-started

[13] Bun Project. “Bun Documentation,” visitado 9 de sep. de 2025. dirección: https://bun.sh/docs

[14] andria_girl. “Using –frozen-lockfile in a CI environment.” Accedido: 11 de septiembre de 2025. dirección: https://stackoverflow.com/questions/73968943/how-to-have-pnpm-install-install-everything-exactly-to-the-specs-of-the-pnpm-l

[15] Angular Team. “What is Angular?” Visitado 8 de sep. de 2025. dirección: https://angular.dev/overview

[16] Angular Team. “HTTP Client — Overview,” visitado 8 de sep. de 2025. dirección: https://angular.dev/guide/http

[17] Angular Team. “Routing — Overview,” visitado 8 de sep. de 2025. dirección: https://angular.dev/guide/routing

[18] Angular Team. “Announcing Angular v20,” visitado 8 de sep. de 2025. dirección: https://blog.angular.dev/announcing-angular-v20-b5c9c06cf301

[19] F. Rappl, *Modern Frontend Development with Node.js: A compendium for modern JavaScript web development within the Node.js ecosystem*. Packt Publishing, 2022, ISBN: 9781804617380. dirección: https://books.google.com.ec/books?id=HyqdEAAAQBAJ

[20] Angular Team. “CLI Reference,” visitado 8 de sep. de 2025. dirección: https://angular.dev/cli

[21] Angular Team. “Setting up the local environment and workspace,” visitado 8 de sep. de 2025. dirección: https://angular.dev/tools/cli/setup-local

[22] Angular Team. “Angular Coding Style Guide,” visitado 8 de sep. de 2025. dirección: https://angular.dev/style-guide

[23] T. Deschryver, *My recommendations to configure Visual Studio Code for Angular Development*, https://timdeschryver.dev/blog/my-recommendations-to-configure-visual-studio-code-for-angular-development, Consultado: 18 septiembre 2024, 2024.

[24] B. Community, *Prettier vs ESLint: Choosing the Right Tool for Code Quality*, https://betterstack.com/community/guides/scaling-nodejs/prettier-vs-eslint/, Consultado: 17 abril 2025, 2025.

[25] A. Thalhammer, *NG Best Practices: Prettier & ESLint*, https://www.angulararchitects.io/blog/best-practices-prettier-eslint/, Consultado: 6 junio 2025, 2025.

[26] OpenJS Foundation. “ESLint - Pluggable JavaScript Linter,” visitado 11 de sep. de 2025. dirección: https://eslint.org

[27] Prettier Contributors. “Prettier - Opinionated Code Formatter,” visitado 11 de sep. de 2025. dirección: https://prettier.io

[28] Angular Team. “Forms in Angular.” Guía oficial: enfoques de formularios reactivos y basados en plantillas («template-driven») en Angular, visitado 11 de sep. de 2025. dirección: https://angular.dev/guide/forms

[29] Angular Team. “Introduction to Angular concepts (Architecture),” visitado 8 de sep. de 2025. dirección: https://angular.io/guide/architecture

[30] ReactiveX. “RxJS - Observables,” visitado 10 de sep. de 2025. dirección: https://rxjs.dev/guide/observable

[31] Angular Team. “Intercepting requests and responses,” visitado 8 de sep. de 2025. dirección: https://angular.dev/guide/http/interceptors

[32] Angular Team. “Angular Documentation,” visitado 9 de sep. de 2025. dirección: https://angular.dev

[33] Angular Team. “Announcing Angular v17,” visitado 11 de sep. de 2025. dirección: https://blog.angular.dev/announcing-angular-v17-5d9730f6f5c4

[34] Webpack Contributors. “Hot Module Replacement Guide,” visitado 9 de sep. de 2025. dirección: https://webpack.js.org/concepts/hot-module-replacement/

[35] *REST: A Pragmatic Introduction to the Web’s Architecture*, Video presentación hospedada en InfoQ, Disponible en https://www.infoq.com/presentations/qcon-tilkov-rest-intro/, consultado el 11 de septiembre de 2025, QCon London, 2009.

[36] Angular Documentation Team. “Providing Services in Angular,” visitado 10 de sep. de 2025. dirección: https://angular.dev/guide/architecture-services

[37] Angular Team. “Essentials: Components (Composition with components),” visitado 10 de sep. de 2025. dirección: https://angular.dev/essentials/components

[38] Angular Team, *Advanced component configuration*, https://angular.dev/guide/components/advanced-configuration, Sección Çhange detection strategy”. Accedido: 11 de septiembre de 2025, 2025.

[39] RxJS Project, *RxJS Documentation*, https://rxjs.dev/guide/overview, Accedido: 11 de septiembre de 2025, 2025.

[40] Moon Technolabs. “A Complete Guide to Data Binding in Angular,” visitado 10 de sep. de 2025. dirección: https://www.moontechnolabs.com/blog/data-binding-in-angular/

[41] Angular Minds. “Guide to Structural Directives in Angular,” visitado 10 de sep. de 2025. dirección: https://www.angularminds.com/blog/structural-directives-in-angular

[42] kamlesh90. “Difference Between REST API and RESTful API.” Accedido: 11 de septiembre de 2025. dirección: https://medium.com/@kamlesh90/difference-between-rest-api-and-restful-api-925dd42a0d1c

[43] Radixweb Blog. “REST vs RESTful APIs: Comparing APIs from a ...” Accedido: 11 de septiembre de 2025. dirección: https://radixweb.com/blog/rest-vs-restful-api

[44] GeeksforGeeks. “Difference Between REST API and RESTful API.” Accedido: 11 de septiembre de 2025. dirección: https://www.geeksforgeeks.org/blogs/know-the-difference-between-rest-api-and-restful-api/

[45] A. Norman. “What’s the difference between REST & RESTful?” Accedido: 11 de septiembre de 2025. dirección: https://stackoverflow.com/questions/1568834/whats-the-difference-between-rest-restful

[46] C. Richardson, *Microservices Patterns: With examples in Java*. Manning Publications, 2018, ISBN: 9781638356325.

[47] Angular Team. “Router: Preventing Unauthorized Access with Route Guards,” visitado 10 de sep. de 2025. dirección: https://angular.dev/guide/router-tutorial-toh#guards

[48] Angular Team. “Angular Security Best Practices,” visitado 10 de sep. de 2025. dirección: https://angular.dev/guide/security

[49] A. Blog. “Angular Route Guards: Protecting Routes in Angular Applications,” visitado 10 de sep. de 2025. dirección: https://auth0.com/blog/angular-route-guards

[50] CACM Staff, “React: Facebook’s Functional Turn on Writing JavaScript,” *Communications of the ACM*, vol. 59, n.º 12, págs. 56-62, 2016. DOI: 10.1145/2980991 dirección: https://cacm.acm.org/practice/react/

[51] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx y W. De Meuter, “A Survey on Reactive Programming,” *ACM Computing Surveys*, vol. 45, n.º 4, págs. 1-34, 2013. DOI: 10.1145/2501654.2501666

[52] I. Malavolta, K. Chinnappan, L. Jasmontas, S. Gupta y K. A. Karam Soltany, “Evaluating the Impact of Caching on the Energy Consumption and Performance of Progressive Web Apps,” en *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft ’20)*, New York, NY, USA: ACM, 2020, págs. 109-119. DOI: 10.1145/3387905.3388593