

Stack Tecnológico

MVC + Capas, Symfony PHP, SvelteJS, SQLite y WebSocket

Adrián Ruiz, Juan Galvis y Miguel Márquez



Contenido

1. Arquitectura MVC + Capas

2. SvelteJS Framework

3. Base de Datos SQLite

4. Protocolo WebSocket

5. Matrices de Análisis

6. Ejemplo práctico

Arquitectura MVC + Capas

Patrón Arquitectónico Fundamental

El sistema se organiza en capas que agrupan responsabilidades similares (por ejemplo: presentación, lógica de negocio y acceso a datos) y, dentro de la capa de presentación, se aplica el patrón MVC.



Vista (View)

Capa de presentación que maneja interfaces de usuario web y móviles, vistas y plantillas.



Controlador (Controller)

Divide claramente la aplicación en capas con roles específicos, evitando dependencias cruzadas.



Modelo (Model)

Núcleo del sistema que organiza en presentación, aplicación, dominio y datos.

Arquitectura en Capas Detallada

Organización Estructural

La arquitectura en capas separa responsabilidades en niveles jerárquicos: presentación, aplicación, dominio y persistencia, facilitando mantenimiento y pruebas.

Capa de Presentación

- Interfaces de usuario web y móviles
- Controladores que manejan peticiones HTTP
- Vistas y plantillas para renderizado

Capa de Aplicación

- Servicios que coordinan casos de uso
- Lógica de flujo y orquestación
- Validación y transformación de datos

Capa de Dominio

Lógica de negocio pura y reglas empresariales.

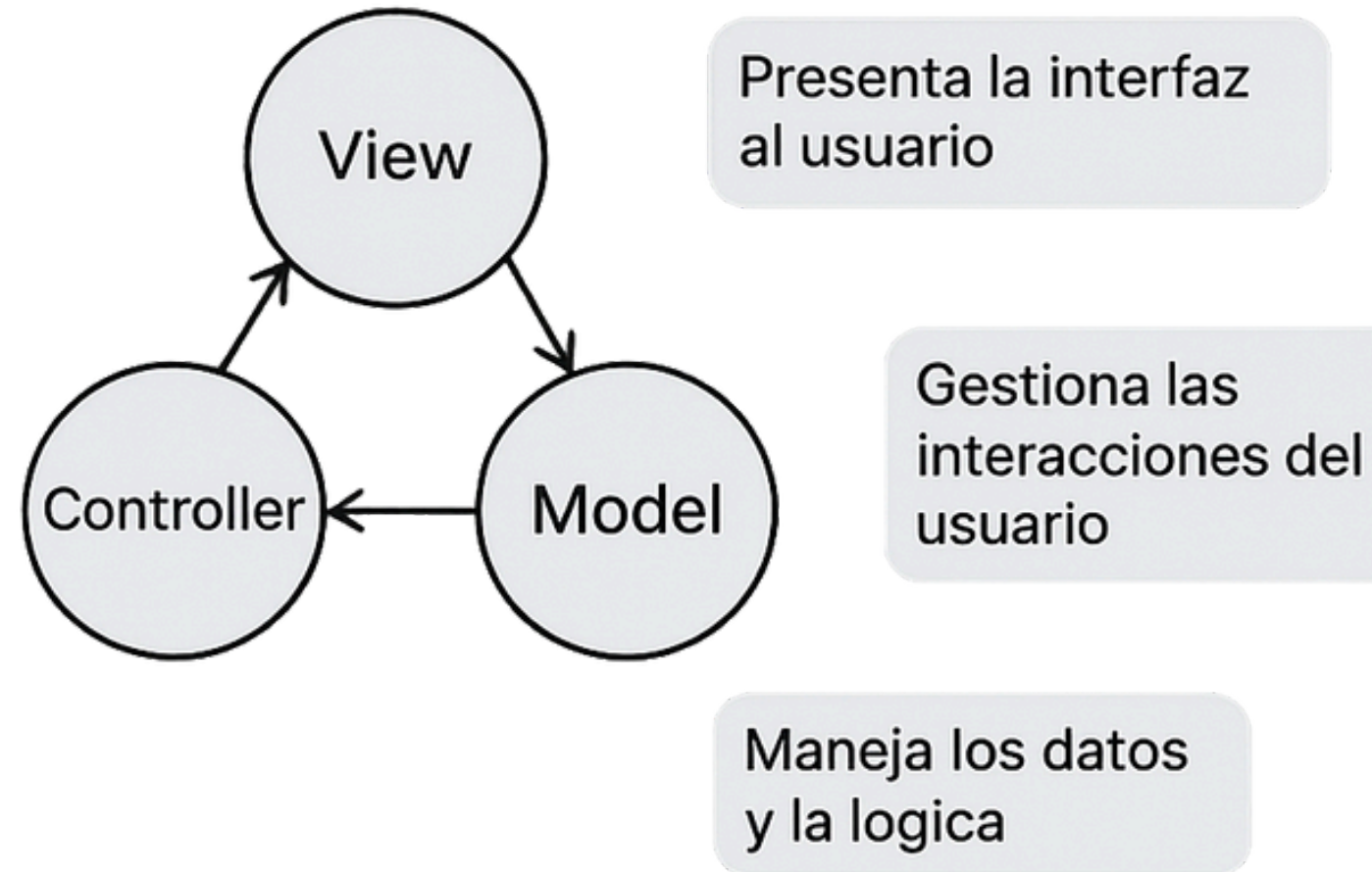
Capa de Persistencia

Acceso a bases de datos y sistemas externos.

Capa de Infraestructura

Implementaciones técnicas y configuraciones.

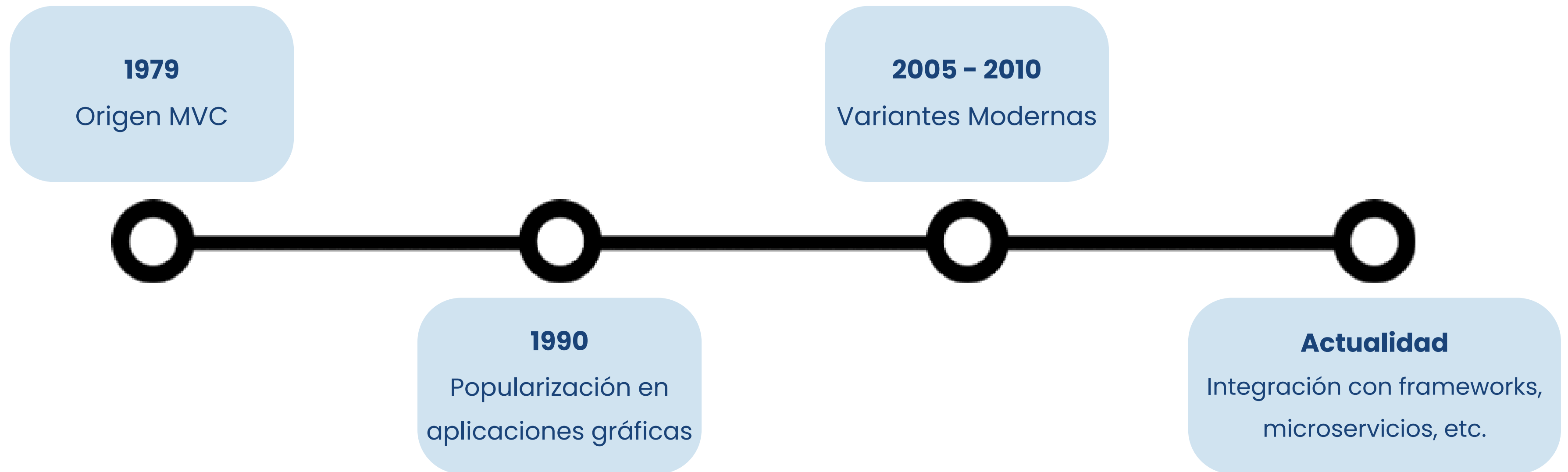
Capa de Presentación



Capa de Negocio

Capa de Datos

Historia y Evolución



Variantes MVC

MVP (Model View Presenter)

Separa la lógica de presentación que actúa entre vista y modelo. Usado en .NET, Android y WinForms.

MVT (Model View Template)

Adaptación usada en Django. La vista gestiona la lógica y el template muestra los datos.

MVA (Model View Adapter)

Introduce un adaptador que traduce datos entre vista y modelo (útil en sistemas complejos o multi-plataforma).

HMVC (Hierarchical MVC)

Divide el sistema en sub-módulos MVC independientes para mejorar la modularidad (usado en PHP y Android).

MVVM (Model View ViewModel)

Propuesto por Microsoft para WPF y Silverlight. Permite “data binding” entre vista y lógica mediante ViewModels. Inspiró frameworks modernos como Angular, React y Vue.

Beneficios de MVC + Capas

Separación de Responsabilidades

Cada componente tiene un rol específico, facilitando el desarrollo en equipos y la especialización.



Facilidad de Mantenimiento

Cambios en una capa no afectan otras, permitiendo evolución independiente del código.



Facilidad de Testing

Cada capa puede probarse de forma aislada con mocks y stubs apropiados.



Escalabilidad Modular

Permite escalar componentes específicos según las necesidades de carga y rendimiento.

Casos de Uso de MVC + Capas

La arquitectura MVC + Capas es especialmente efectiva en proyectos que requieren organización estructural, mantenibilidad a largo plazo y desarrollo colaborativo.

Situación	Descripción
Aplicaciones empresariales	Ideal para sistemas con muchas funcionalidades, múltiples usuarios y operaciones complejas (por ejemplo, plataformas bancarias, educativas o ERP).
Aplicaciones web modernas	Permite separar claramente la lógica de presentación (frontend) y la lógica del negocio (backend). Usado en portales, e-commerce, redes sociales, etc.
Sistemas con múltiples interfaces	Cuando un mismo backend debe atender una app web, una app móvil y una API, las capas ayudan a reutilizar lógica y mantener consistencia.
Proyectos de larga duración	Facilita el mantenimiento, actualizaciones y reemplazo de tecnologías sin reescribir todo el sistema.
Entornos académicos o de formación	Es el patrón más usado para enseñar principios de diseño, modularidad y buenas prácticas de desarrollo.
Sistemas con equipos distribuidos	Al dividir el software en capas, varios grupos (frontend, backend, base de datos) pueden trabajar en paralelo sin interferirse.
Aplicaciones con necesidad de pruebas automáticas	Las capas bien definidas permiten crear pruebas unitarias e integradas para cada componente.

Casos de Aplicación de MVC + Capas

La arquitectura MVC + Capas es especialmente efectiva en proyectos que requieren organización estructural, mantenibilidad a largo plazo y desarrollo colaborativo.

Aplicación / Framework	Descripción	Implementación de MVC + Capas
Spring Boot (Java)	Framework empresarial basado en Spring MVC. Organiza el sistema en controladores, servicios y repositorios.	Usa MVC en la capa de presentación y una arquitectura de 3 capas para backend.
Django (Python)	Framework web con patrón MVT (similar a MVC).	Vista controla la lógica, Template maneja presentación y Modelo gestiona datos.
ASP.NET Core (C#)	Framework de Microsoft para web y APIs REST.	Implementa MVC con separación por capas y soporta inyección de dependencias.
Angular (TypeScript)	Framework frontend basado en MVVM.	Usa componentes (vista), servicios (modelo) y binding bidireccional entre ambos.
React (JavaScript)	Librería frontend declarativa inspirada en MVVM.	La vista se actualiza automáticamente cuando cambian los estados del modelo.
Laravel (PHP)	Framework MVC modular con arquitectura en capas.	Divide el código en controladores, modelos, vistas y servicios reutilizables.
Android (Kotlin/Java)	Usa MVVM y arquitectura en capas (UI, dominio, datos).	Ideal para separar la lógica de presentación del acceso a APIs o BD local.

Symfony PHP

Framework PHP

Symfony es un framework web en PHP, gratuito y de código abierto, que facilita crear aplicaciones modernas, organizadas y fáciles de mantener mediante el patrón MVC. Su estructura modular y herramientas avanzadas lo hacen flexible, seguro y eficiente para distintos tipos de proyectos



Componentes Modulares

Sistema de bundles independientes con alta modularidad, compatible con PSR y estándares de PHP.



Inyección Dependencias

Contenedor avanzado que permite aislar lógica y facilitar pruebas unitarias.



Seguridad Robusta

Sistema completo de autenticación, CSFR y encriptación integrados nativamente.

Evolución de Symfony



2005: Nacimiento del Framework

Fabien Potencier crea Symfony como framework PHP para proyectos web empresariales con arquitectura MVC.



2011: Componentes Independientes

Symfony 2 introduce componentes desacoplados, siendo adoptados por otros frameworks como Laravel y Drupal.



2017: Symfony 4 y Flex

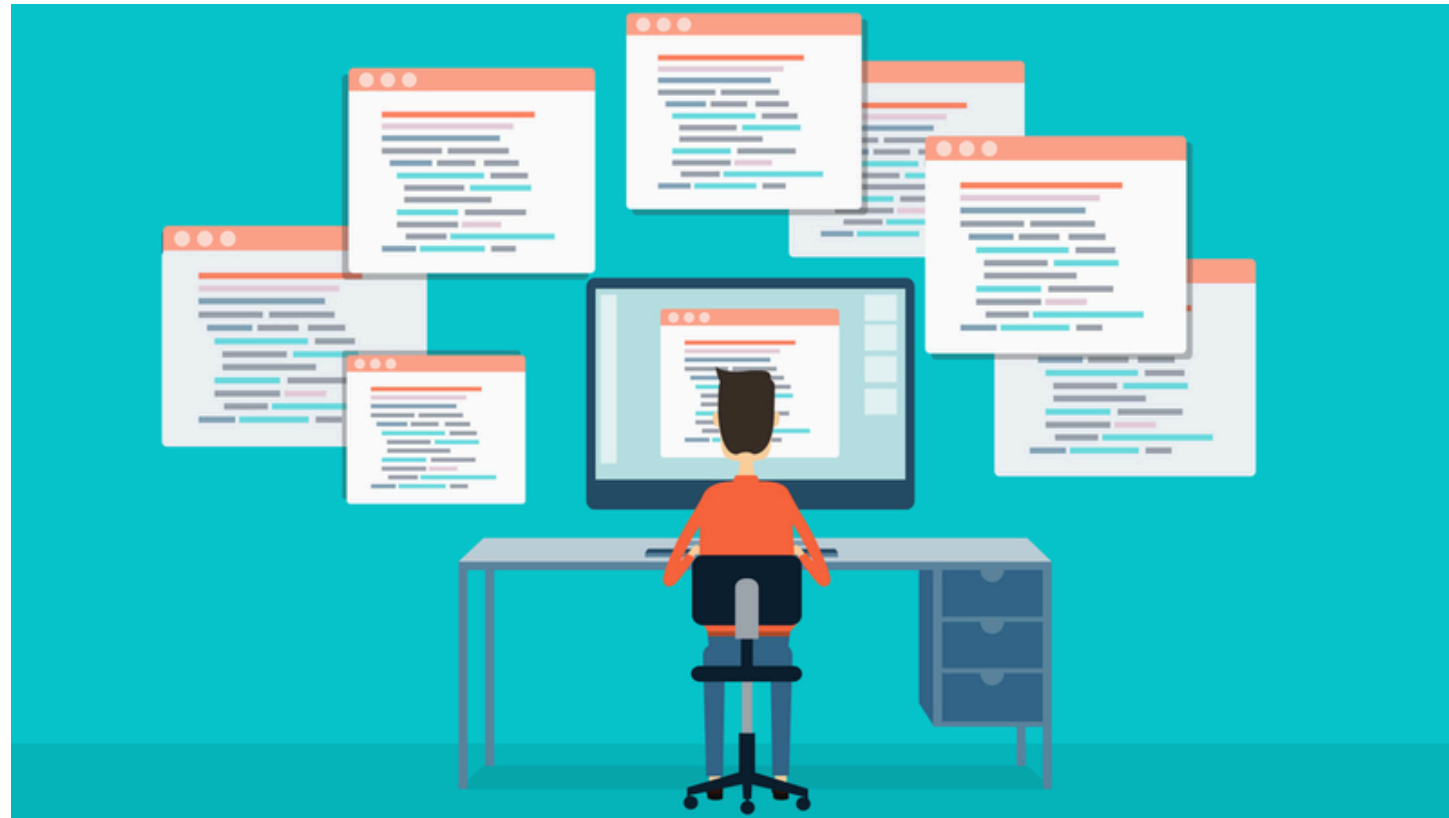
Nueva estructura de proyecto con autocarga automática, mejoras de performance y developer experience optimizada.



2019–2024: Modernización Continua

Symfony 5 y 6 añaden compatibilidad con PHP 8+, mejoras de tipado y herramientas de desarrollo avanzadas.

Ventajas y Desventajas de Symfony

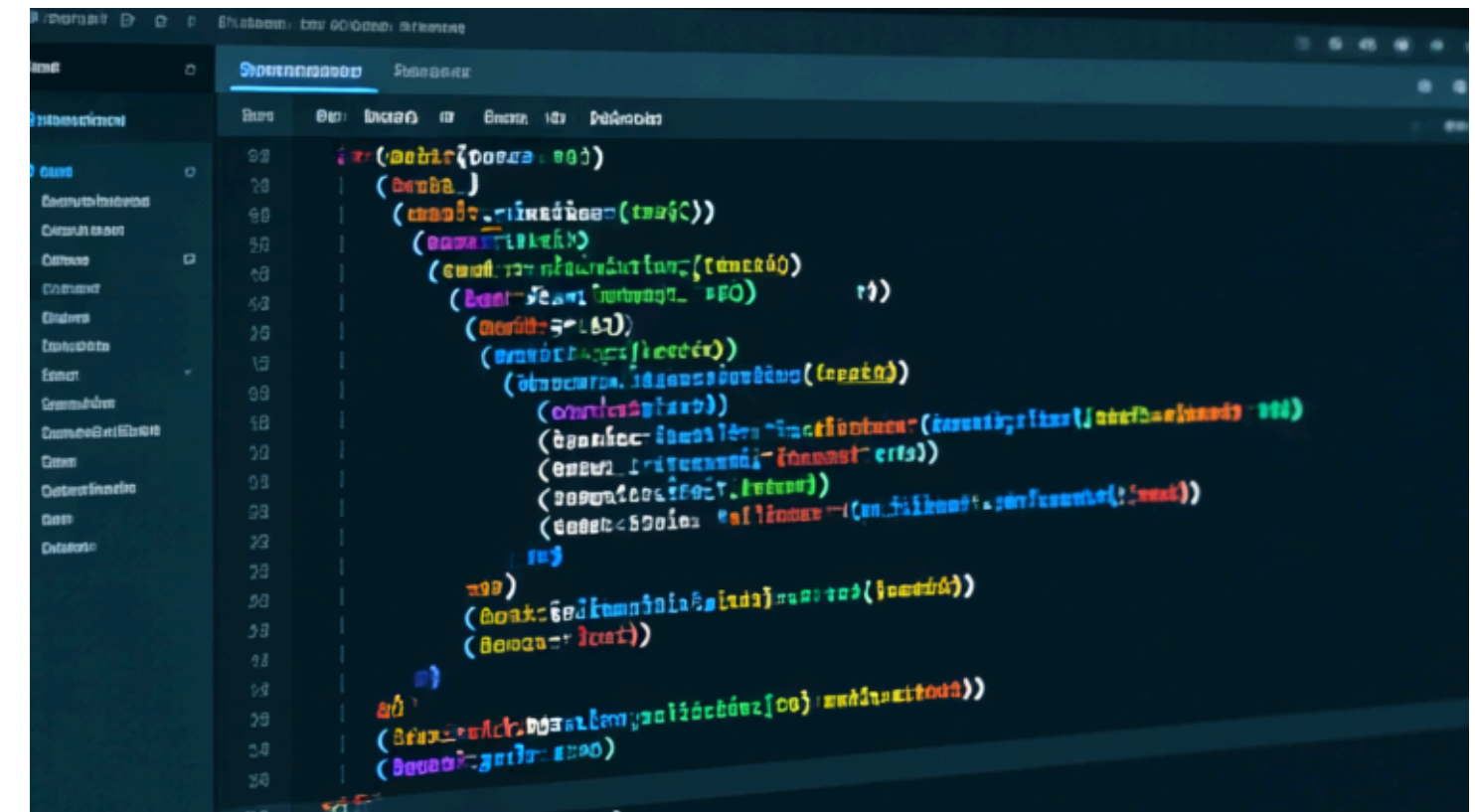


Consideraciones y Desafíos

- Curva de aprendizaje pronunciada para desarrolladores junior
- Configuración inicial compleja en proyectos grandes
- Overhead de performance comparado con microframeworks

Ventajas Principales

- Arquitectura madura y estable probada en producción
- Comunidad activa con documentación extensa y soporte
- Compatibilidad con estándares PSR y mejores prácticas PHP



Casos de Uso de Symfony

Situación / Escenario	Descripción
Plataformas web de larga duración	Gracias a sus versiones con soporte a largo plazo (LTS) y su estabilidad, Symfony es perfecto para proyectos que necesitan mantenimiento continuo durante años.
APIs REST y microservicios	Symfony permite crear APIs estables y seguras usando componentes como API Platform. Su soporte para JSON, controladores y autenticación lo hace ideal para conectar aplicaciones móviles o frontends modernos.
Portales institucionales o educativos	Su sistema de roles, formularios y seguridad permite desarrollar portales de universidades, gobiernos o empresas que gestionen usuarios y contenidos con control de acceso.
E-commerce personalizados	En tiendas en línea con lógica de negocio compleja (descuentos, inventarios, reportes, pasarelas de pago), Symfony ofrece control total sobre la arquitectura y permite integrar sistemas externos como ERP o CRM.
Sistemas internos (Intranets)	Muchas empresas utilizan Symfony para crear intranets corporativas seguras, con autenticación de empleados, carga de archivos, gestión documental y reportes.
Plataformas SaaS (Software como servicio)	Symfony soporta multi-tenant (varios clientes en una sola aplicación) y facilita la administración de suscripciones, autenticación y escalabilidad en la nube.
Aplicaciones con altos requerimientos de seguridad	Sus mecanismos integrados de cifrado, validación y control de acceso lo hacen ideal para proyectos donde los datos deben protegerse, como bancos, hospitales o sistemas legales.
Proyectos modulares o distribuidos	Symfony permite desarrollar módulos independientes (bundles) que pueden integrarse fácilmente o compartirse entre varios proyectos. Esto acelera el desarrollo de sistemas grandes y en equipo.
Integración con otros frameworks o CMS	Muchas plataformas como Drupal, Magento o Laravel utilizan componentes de Symfony, lo que permite integrar nuevas funciones o migrar proyectos sin perder compatibilidad.

Casos de Aplicación de Symfony

Nombre / Empresa	Uso / contexto	Descripción
Upply	Plataforma de transporte / logística digital	Decidieron mantenerse con PHP + Symfony a pesar de considerar migrar a lenguajes “más modernos”, debido al ecosistema y experiencia del equipo.
Drupal (versión 8 en adelante)	CMS muy difundido	Drupal integró los componentes de Symfony para mejorar la modularidad y modernidad de su arquitectura.
Grandes portales y sitios	Sitios de contenido, plataformas institucionales	Usan Symfony para manejar tráfico elevado, módulos personalizados, integración con otros sistemas.
E-commerce sofisticados	Tiendas en línea que requieren lógica compleja, personalización, escalabilidad	Symfony es usado cuando el e-commerce debe ser extensible, integrado con otros sistemas (ERP, CRM).
Aplicaciones internas / intranet corporativa	Portales internos, herramientas de gestión empresarial	Symfony es adecuado porque permite organizar funciones complejas con control de seguridad, roles, flujos de negocio.

SvelteJS

Compilador de Componentes

SvelteJS es un framework moderno que compila componentes a JavaScript optimizado, ofreciendo mejor rendimiento y menor tamaño de bundle que frameworks tradicionales.



Bundle Ligero

Compilación que genera código JavaScript mínimo sin framework runtime pesado.



Sintaxis Intuitiva

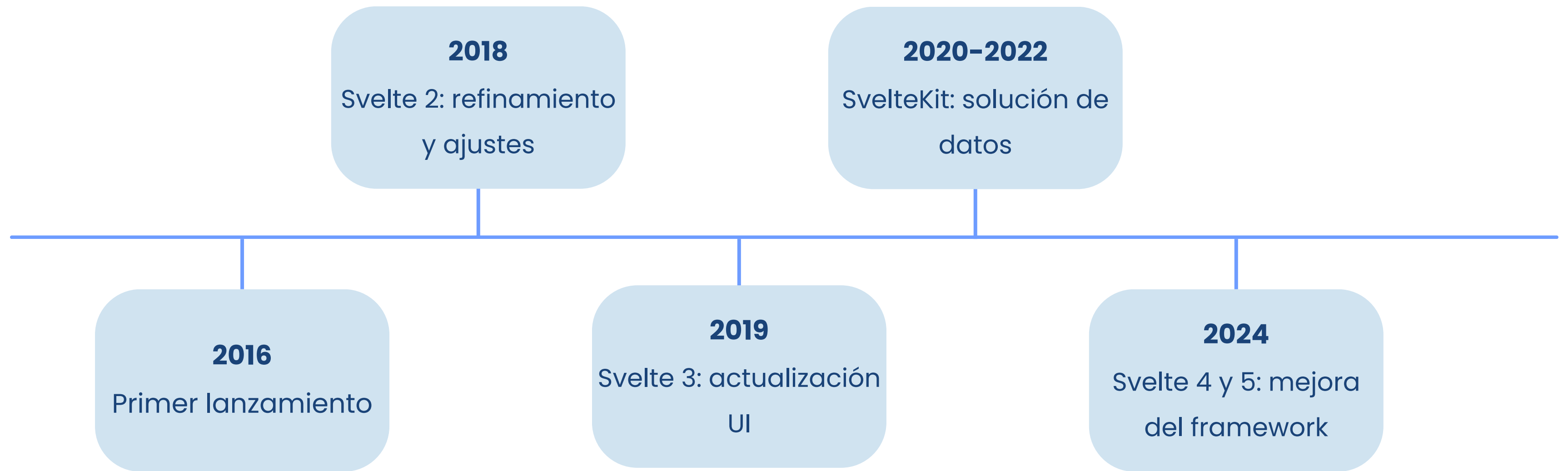
Desarrollo simplificado con menos boilerplate y curva de aprendizaje suave.



Performance Nativa

Reactividad compilada que elimina Virtual DOM y optimiza actualizaciones.

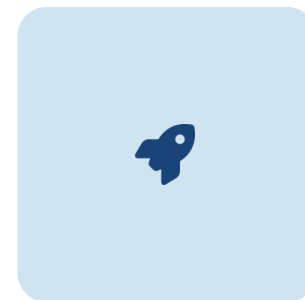
Historia y Evolución de SvelteJS



Ventajas de SvelteJS

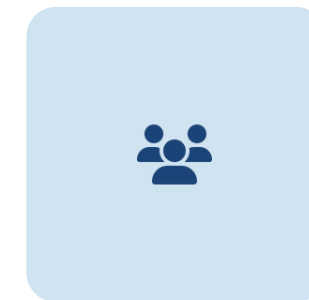
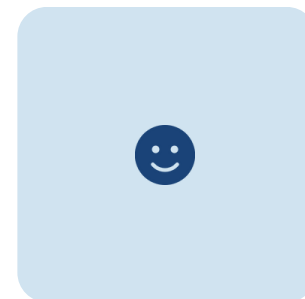
Carga rápida / rendimiento

Al compilar y eliminar runtime, la aplicación que llega al navegador es más ligera y responde más rápido.



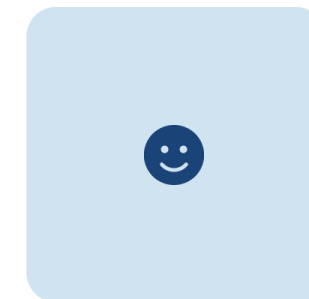
Fácil de aprender

Su sintaxis es cercana a HTML/JS, sin mucha abstracción extra, lo que acelera su curva de adopción.



Menor tamaño de bundle

Al no incluir el peso del framework en runtime, el paquete final es más pequeño y eficiente.



Reactivo de forma natural

Los cambios en variables disparan actualizaciones automáticas de UI sin librerías adicionales.

Casos de Uso de SvelteJS

Escenario	Uso
Prototipos rápidos / MVPs	La simplicidad y velocidad de desarrollo permiten validar ideas con poco esfuerzo.
Aplicaciones web con alta performance	Ideal cuando se necesita que la UI responda rápido, con poca latencia y animaciones fluidas.
Dispositivos con recursos limitados	En clientes con poca CPU o conexiones lentas, un bundle ligero marca la diferencia.
Dashboards y visualización de datos	La reactividad y animaciones nativas ayudan a actualizar gráficos e interfaces dinámicas eficientemente.
Aplicaciones multipágina o híbridas	Cuando necesitas rutas y carga de datos, SvelteKit complementa perfectamente el framework.
Interfaces interactivas aisladas en apps grandes	Puedes usar Svelte para ciertos módulos o widgets dentro de una aplicación más grande.

Casos de Aplicación de SvelteJS



Empresas Pioneras

- Square: interfaces móviles y páginas interactivas optimizadas
- Grammarly: editor web con interactividad mejorada
- The New York Times: secciones interactivas del sitio web



Corporaciones Globales

- IKEA: componentes e-commerce específicos y eficientes
- Yahoo: módulos de interfaz de usuario optimizados
- Spotify: elementos interactivos de plataforma musical



Comunidad Activa

- Portales web modernos con diseño responsivo
- Dashboards empresariales internos y ligeros
- Aplicaciones web especializadas de alta eficiencia

SQLite: Base de Datos

Gestión de bases de datos relacional

SQLite es un RDBMS ligero que almacena toda la base de datos en un único archivo físico, eliminando la necesidad de servidor independiente.

Características Principales

- Implementa estándar SQL-92 casi completo
- Transacciones ACID para integridad de datos
- Tamaño inferior a 1 MB total

Aplicaciones Ideales

- Aplicaciones móviles Android e iOS
- Software de escritorio y embebido
- Navegadores web y sistemas IoT

Ligera

Menos de 1 MB de tamaño.

Portátil

Funciona en cualquier sistema operativo.

Confiable

Probada por millones de aplicaciones.

Historia y Evolución de SQLite

2000: Creación de SQLite

Desarrollado por D. Richard Hipp con el objetivo de crear una base de datos pequeña y fácil de integrar.



2001–2004: Primeras Versiones Estables

Se añade soporte para transacciones y sentencias complejas SQL. Gana popularidad en entornos UNIX y sistemas embebidos.



2005: Adopción por Proyectos Grandes

Firefox y otros navegadores comienzan a usar SQLite como motor de almacenamiento local.



2008: Integración en Android e iOS

Se convierte en la base de datos por defecto para dispositivos móviles, impulsando su expansión global.

SQLite: Ventajas vs Desventajas

Ventajas

Beneficios técnicos y operacionales de SQLite según análisis detallado del documento de referencia.

- Sin necesidad servidor: único archivo elimina configuración y procesos administración
- Liviana rápida: tamaño permite ejecución eficiente dispositivos pocos recursos
- Fácil usar integrar: incorpora directamente cualquier lenguaje sin configuraciones extra

Desventajas

Limitaciones técnicas específicas de SQLite según análisis crítico del documento fuente.

- No multiusuario concurrente: solo una escritura vez, no ideal muchos usuarios
- Falta funciones avanzadas: no herramientas administración monitoreo como MySQL PostgreSQL
- No apta entornos distribuidos: sin replicación clustering nativo limita gran escala

Casos de Uso SQLite

Escenario	Descripción
Aplicaciones móviles	Android e iOS utilizan SQLite como base de datos local por su ligereza y bajo consumo.
Software de escritorio	Ideal para aplicaciones pequeñas que necesitan guardar información local (por ejemplo, historial, configuración o sesiones).
Proyectos educativos o de prueba	Perfecta para aprender SQL o probar prototipos sin instalar servidores externos.
Aplicaciones embebidas / IoT	Muy usada en dispositivos inteligentes, autos y electrodomésticos, donde se requiere almacenamiento pequeño y confiable.
Navegadores web	Firefox, Chrome y Safari usan SQLite para gestionar historial, cookies y marcadores.
Aplicaciones empresariales ligeras	Usada en sistemas internos que no requieren múltiples usuarios concurrentes.
Copias locales o caché de sistemas grandes	Puede servir como almacenamiento temporal o caché local de una base de datos más grande.

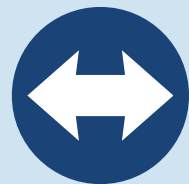
Casos de Aplicación SQLite

Proyecto / Empresa	Uso de SQLite	Relevancia
Android OS (Google)	Base de datos predeterminada para todas las aplicaciones móviles.	Millones de apps en Android usan SQLite internamente.
Apple iOS / macOS	Utilizada en Core Data y apps del sistema (Mensajes, Notas, Contactos).	Garantiza almacenamiento local eficiente en todos los dispositivos Apple.
Mozilla Firefox	Almacena historial, contraseñas y cookies.	Proporciona un rendimiento rápido y estable para millones de usuarios diarios.
Google Chrome	Usa múltiples bases SQLite para historial, pestañas y marcadores.	Permite persistencia rápida sin ralentizar el navegador.
Dropbox	Emplea SQLite para manejar sincronización local de archivos antes de subirlos a la nube.	Aumenta la confiabilidad y evita pérdida de datos.
Skype y WhatsApp Desktop	Guardan mensajes y configuraciones localmente usando SQLite.	Permite funcionamiento offline y sincronización posterior.
Tesla / Automotriz	Usan SQLite para registrar datos de sensores e información del sistema a bordo.	Fiabilidad y bajo consumo en entornos embebidos.

WebSocket

Protocolo de comunicación bidireccional

Permite establecer una conexión continua entre cliente y servidor. A diferencia de HTTP request-response, mantiene un canal abierto para enviar y recibir datos en tiempo real.



Comunicación bidireccional

Cliente y servidor pueden enviar datos en cualquier momento sin solicitudes repetidas.



Conexión Persistente

Una vez establecida, permanece abierta hasta que alguna parte la cierre.



Bajo consumo Banda

Reduce sobrecarga HTTP enviando encabezados solo una vez durante el handshake inicial.

Características Técnicas de WebSocket

Característica	Descripción Técnica	Ventaja Clave	Consideración
Comunicación bidireccional	Cliente y servidor envían datos cualquier momento	Sin solicitudes repetidas	Requiere manejo estados
Conexión persistente	Permanece abierta hasta cierre partes	Evita overhead crear conexiones	Consume recursos servidor
Bajo consumo ancho banda	Encabezados solo en handshake inicial	Reduce sobrecarga HTTP	Conexiones persistentes memoria
Basado en TCP	Garantiza confiabilidad orden entrega	Mensajes completos ordenados	Dependiente calidad red
Soporte binarios y texto	Transmite JSON e imágenes/audio	Flexibilidad tipos datos	Requiere manejo diferenciado
Seguridad integrada WSS	Cifrado TLS igual HTTPS	Comunicación segura	Configuración adicional necesaria

Historia y Evolución de WebSocket

Evolución del protocolo WebSocket desde su concepto inicial hasta su integración con tecnologías modernas

Año	Descripción	Importancia
2008	Creación del concepto por Ian Hickson	Se propuso dentro del proyecto WHATWG como solución a la falta de comunicación bidireccional en la web
2009	Primer prototipo implementado en navegadores	Se realizaron las primeras pruebas de conexión persistente en Chrome y Firefox
2011	Publicación del estándar oficial RFC 6455	El IETF publica la especificación formal del protocolo WebSocket, marcando su adopción generalizada
2013–2015	Soporte en servidores y frameworks	Node.js, Java EE, .NET y otros incorporan librerías oficiales para WebSocket
2016–2020	Expansión en aplicaciones modernas	Se vuelve esencial para chats, notificaciones, streaming y dashboards en tiempo real
2021–2024	Integración con nuevas tecnologías	Se combina con WebRTC, GraphQL Subscriptions y microservicios, fortaleciendo su rol en la web moderna

WebSocket: Ventajas vs Desventajas

Ventajas

Beneficios específicos del protocolo WebSocket

- Comunicación en tiempo real: enviar y recibir datos instantáneamente sin recargar página
- Reducción de latencia: conexión persistente elimina retardo de establecer nuevas solicitudes HTTP
- Ahorro de recursos: menor consumo ancho de banda y CPU gracias reutilización canal TCP

Desventajas

Limitaciones técnicas específicas de WebSocket

- Complejidad en escalabilidad: conexiones persistentes pueden consumir muchos recursos del servidor
- Mantenimiento de conexión: requiere manejar reconexiones y detección errores manualmente
- Compatibilidad con proxies y firewalls: algunos entornos bloquean conexiones WebSocket

Casos de Uso de WebSocket

Escenario	Descripción
Chats en tiempo real	Los mensajes se envían y reciben instantáneamente sin recargar la página
Juegos en línea	Permite la comunicación continua entre jugadores y servidores sin interrupciones
Notificaciones en vivo	Ideal para alertas, actualizaciones de estado o mensajes emergentes en aplicaciones web
Paneles de monitoreo (dashboards)	Muestra datos actualizados constantemente, como sensores o métricas de sistemas
Aplicaciones financieras	Facilita la actualización de cotizaciones o movimientos en tiempo real
Aplicaciones colaborativas	Permite a varios usuarios editar o interactuar simultáneamente (como Google Docs)
Transmisiones en vivo / streaming	Optimiza la entrega continua de datos de audio o video
IoT	Conecta dispositivos inteligentes para enviar datos y recibir comandos de manera instantánea

WebSocket en Aplicaciones Empresariales

Proyecto/Empresa	Uso de WebSocket	Relevancia o Beneficio
Slack	Comunicación en tiempo real entre usuarios y canales	Permite mensajería instantánea con sincronización en todos los dispositivos
Trello	Actualiza tableros de tareas sin recargar la página	Ofrece colaboración fluida entre equipos
Binance / Coinbase	Transmisión en vivo de precios y órdenes de mercado	Reduce la latencia en datos financieros
Facebook Messenger / WhatsApp Web	Mantiene las conversaciones actualizadas en tiempo real	Brinda experiencia continua sin recargas
Google Docs	Permite edición colaborativa simultánea	Varios usuarios pueden trabajar sobre el mismo documento sin conflictos
Spotify Web Player	Usa WebSocket para sincronizar canciones y estados de reproducción	Garantiza actualización instantánea entre cliente y servidor
TradingView	Envía gráficos y valores financieros en vivo	Imprescindible para análisis bursátil en tiempo real

Relación Entre Los Temas Asignados

Stack Tecnológico Completo

El conjunto MVC + Capas, Symfony PHP, SvelteJS, SQLite y WebSocket forma un stack equilibrado que combina herramientas tradicionales con tecnologías modernas orientadas a eficiencia e interactividad.



Equilibrio Tradicional-Moderno

Symfony y MVC representan base estable probada, mientras SvelteJS aporta modernidad reactiva.



Eficiencia Práctica

Stack prioriza eficiencia y aplicabilidad sobre popularidad, ideal para proyectos donde rendimiento vale más que tendencias.



Adopción Específica

Poco común en grandes corporaciones, muy usado en entornos académicos, startups y proyectos personalizados.

Principios SOLID en el Stack

Principio	MVC + Capas	Symfony – PHP	SvelteJS – JS	SQLite	WebSocket
S – Single Responsibility	Cada capa asume un único rol (presentación, aplicación, datos) y dentro de presentación, MVC separa Vista/Controlador/Modelo;	Controladores delgados y Servicios/Repositorios/Forms con tareas acotadas; DI permite aislar lógica y pruebas.	SRP depende de dividir componentes y extraer lógica a stores/servicios; los componentes tienden a crecer si no se segmentan.	Capa DAO/Repositorio encapsula exclusivamente persistencia; el dominio no conoce SQL ni detalles de archivo.	Conviene separar conexión, protocolo de mensajes y handlers por tópico; si se centraliza todo en un único 'socket service' se viola SRP.
O – Open/Closed	Se agregan casos de uso o capas sin modificar las existentes mediante interfaces/puertos; extensible por composición.	Extensión por eventos/listeners y decoradores de servicios; nuevos bundles añaden capacidades sin tocar código base.	Extensión vía props/slots/stores; si el contrato del componente no está claro, futuras extensiones fuerzan cambios internos.	Esquemas estables se amplían con vistas/queries/DAOs; cambios de modelo pueden requerir migraciones que modifican estructuras.	Nuevos tipos de mensajes/eventos se agregan sin romper los existentes si el protocolo está versionado.
L – Liskov Substitution	Servicios/Repositorios con interfaces permiten intercambiar implementaciones (p.ej., mock vs real) sin alterar clientes.	Contratos definidos por interfaces y tipos; DI inyecta sustitutos (mocks/stubs) manteniendo el comportamiento esperado.	Sustitución depende de respetar contratos de props/eventos; supuestos implícitos de estado rompen LSP.	Backends de almacenamiento pueden cambiarse si el repositorio mantiene el mismo contrato.	Se puede sustituir el transporte (WS↔SSE/HTTP polling) si la interfaz de mensajería se abstrae correctamente.
I – Interface Segregation	Capas exponen interfaces específicas (puertos de aplicación); los controladores no dependen de detalles de persistencia.	Servicios y Repositorios con interfaces pequeñas y enfocadas; validadores/forms desacoplados.	Segmentar props/eventos mínimos por componente y stores especializados; evitar componentes 'omnívoros'.	Repositorios con métodos focalizados por agregado; evitar 'mega DAO' con demasiadas operaciones.	Separar canales/rooms y handlers por tópico; una interfaz gigante de eventos viola ISP.
D – Dependency Inversion	Capas altas dependen de abstracciones (puertos/servicios); infraestructura se inyecta como implementación concreta.	Contenedor de DI primero: controladores dependen de interfaces; infraestructura se resuelve por configuración.	Invertir dependencias moviendo fetch/WS a servicios y proveyéndolos a componentes; sin esto, la UI queda acoplada.	El dominio depende de repositorios abstractos; la implementación SQLite vive en infraestructura.	UI/dominio dependen de una abstracción de mensajería; el adaptador concreto (WS) se inyecta.

Atributos de Calidad en el Stack

Atributo de Calidad	MVC + Capas	Symfony – PHP	SvelteJS – JavaScript	SQLite	WebSocket
Adecuación funcional	Estructura bien definida que asegura que cada capa cumpla su función.	Cumple ampliamente por su modularidad y componentes reutilizables.	Proporciona una interfaz reactiva que mejora la precisión funcional de la UI.	Base de datos estable y confiable con soporte completo de SQL.	Permite funcionalidades en tiempo real (mensajería, notificaciones).
Rendimiento / Eficiencia de desempeño	Reduce complejidad y optimiza llamadas, depende de implementación.	Alto rendimiento con caché y optimización, requiere configuración avanzada.	Genera código JS optimizado en compilación, excelente rendimiento.	Acceso directo a disco, muy rápido en entornos locales.	Transmisión continua y eficiente, mínima latencia.
Mantenibilidad	Estructura clara que facilita lectura y modificación del código.	Alta mantenibilidad con DI, bundles y documentación extensa.	Componentes aislados facilitan mantenimiento, aunque comunidad es pequeña.	Liviano y de bajo mantenimiento, pero limitado para proyectos grandes.	Requiere modularizar eventos para evitar código complejo.
Escalabilidad	Escalable dividiendo capas, aunque requiere refactorizaciones.	Altamente escalable por su arquitectura modular.	Escalable en frontend si se segmenta correctamente.	Limitada a pocos usuarios, no apta para grandes volúmenes.	Escalable con clusters, pero complejo de gestionar.
Seguridad	Permite filtros y validaciones en cada capa.	Sistema robusto de autenticación, CSRF y encriptación.	Depende del backend, pero protege frente a XSS.	Seguridad básica; depende del sistema operativo.	Cifrado TLS (WSS) y validación para proteger datos.
Compatibilidad / Interoperabilidad	Fácil de integrar con otros sistemas mediante APIs.	Cumple PSR y estándares PHP-FIG; alta interoperabilidad.	Compatible con cualquier backend REST o WebSocket.	Compatible con múltiples lenguajes mediante SQL estándar.	Intercambia datos entre distintos lenguajes y frameworks.
Usabilidad	Fácil de entender y mantener; estructura clara para nuevos desarrolladores.	Herramientas CLI y profiler mejoran la experiencia del desarrollador.	Excelente experiencia de usuario final con UI fluida y reactiva.	No afecta la interfaz, pero mejora confiabilidad de datos.	Permite interacciones instantáneas y fluidas sin recarga.
Portabilidad	Diseño modular facilita migrar módulos a otros entornos.	Altamente portable en múltiples servidores y sistemas.	Compila a JS estándar compatible con cualquier navegador.	Extremadamente portable; un solo archivo ejecutable en cualquier SO.	Amplio soporte en navegadores y servidores modernos.

Tácticas vs Tecnologías

Táctica / Tema	MVC + Capas	Symfony – PHP	SvelteJS – JavaScript	SQLite	WebSocket
Separación de responsabilidades (mantenibilidad)	Divide claramente la aplicación en capas con roles específicos, evitando dependencias cruzadas.	Refuerza esta separación mediante controladores, servicios y entidades.	Logra separación si los componentes son pequeños y la lógica se extrae a stores.	Permite aislar la capa de persistencia mediante DAOs o repositorios.	Requiere modularizar eventos y listeners para mantener claridad en la lógica.
Uso de caché (rendimiento)	Puede implementar caché a nivel de capa o controlador para reducir tiempos de respuesta.	Ofrece sistemas de caché HTTP y resultados de consultas integrados.	Usa el renderizado reactivo, lo que reduce el trabajo del DOM.	Caché implícito en consultas frecuentes, aunque no nativo.	Reutiliza conexiones persistentes, minimizando overhead en transmisión.
Control de concurrencia (consistencia / rendimiento)	Facilita sincronización por capas; cada capa gestiona su dominio.	Gestiona concurrencia mediante bloqueos de base de datos y control transaccional.	Controla estados compartidos con stores o reactive variables.	Soporta lectura concurrente, escritura exclusiva; ideal para baja concurrencia.	Gestiona múltiples conexiones simultáneas mediante canales o tópicos.
Autenticación y autorización (seguridad)	Se ubica en capa de aplicación para separar lógica de acceso.	Posee componentes nativos de autenticación, roles y cifrado.	Depende del backend, pero puede integrar validación JWT o tokens.	No incluye seguridad interna; depende del sistema que la use.	Usa verificación de origen y tokens para conexiones seguras.
Validación de entrada (seguridad / confiabilidad)	Cada capa puede validar datos antes de pasarlos a la siguiente.	Incluye validadores automáticos y constraints en formularios.	Validación en tiempo real en formularios reactivos.	No valida por sí mismo; validación ocurre en el cliente o aplicación.	Validación de mensajes en el servidor para evitar inyecciones.

Patrones vs Tecnologías

Patrón	MVC + Capas	Symfony – PHP	SvelteJS – JavaScript	SQLite	WebSocket
MVC (Model–View–Controller)	Patrón base: separa la lógica, la vista y los datos, logrando modularidad y mantenibilidad.	Implementado nativamente con controladores, vistas Twig y entidades Doctrine.	No sigue MVC puro, pero usa un modelo reactivo (MVU/MVVM) para separar UI y estado.	No aplica directamente, pero puede participar como modelo de datos.	Se integra como backend MVC que usa WS como canal de comunicación.
Arquitectura en Capas (N-Tier)	Núcleo del estilo: organiza el sistema en presentación, aplicación, dominio y datos.	Adoptada totalmente: Controller → Service → Repository → DB.	Adaptada en frontend con capas de componentes, servicios y red.	Representa la capa de persistencia.	Se implementa como capa de transporte dentro del backend.
Microkernel (Plug-in Architecture)	Facilita ampliaciones por módulos o servicios sin alterar el núcleo.	Soportado por bundles que funcionan como plugins independientes.	Soporta modularidad por componentes y librerías externas.	No aplica; SQLite es un motor compacto.	Puede emplearse para módulos de protocolo o handlers.
Event-Driven Architecture (EDA)	Se usa entre capas para emitir y manejar eventos internos.	Symfony posee EventDispatcher y suscriptores para procesos asíncronos.	Nativo: los componentes reaccionan a eventos del DOM o stores.	Se integra fácilmente en un sistema EDA mayor.	Es el núcleo: los mensajes WS funcionan como eventos distribuidos.
CQRS (Command Query Responsibility Segregation)	Aplica separando controladores o servicios de lectura y escritura.	Doctrine y controladores permiten aplicar CQRS.	Stores para lectura y acciones para escritura implementan este patrón naturalmente.	Puede actuar como modelo de lectura local sincronizado.	WS separa comandos (escritura) y suscripciones (lectura).

Patrones vs Tecnologías

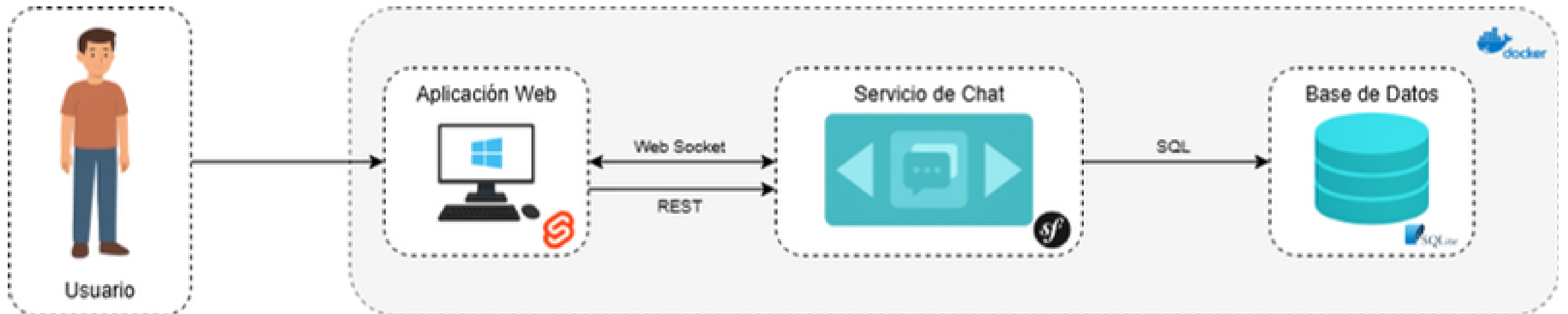
Strategy	Define algoritmos intercambiables por capa (validación, seguridad, lógica de negocio).	Usado para cacheo, serialización, y políticas de autenticación.	Estrategias para renderización o gestión de datos.	Estrategias de consultas según carga o índice.	Estrategias para reconexión o retransmisión de mensajes.
Factory / Abstract Factory	Crea instancias de servicios y controladores de forma flexible.	Utilizado en creación de servicios y entidades Doctrine.	Factories para componentes, stores o servicios reutilizables.	Fábricas de conexiones o consultas parametrizadas.	Fábricas de sockets o handlers por canal.
Adapter	Conecta capas de aplicación con sistemas externos.	Adaptadores para correo, APIs o servicios externos.	Adaptadores de API/WS para desacoplar la lógica de red.	Adaptadores que unifican acceso a SQLite.	Adaptador WS o SSE bajo interfaz común de mensajería.
Facade	Simplifica el uso de subsistemas complejos desde capas superiores.	Services o managers centralizan acceso a módulos del framework.	Servicios que agrupan lógica y llamadas a APIs.	Fachada de persistencia para manejo de conexiones y consultas.	Fachada de tiempo real para enviar, recibir y cerrar conexiones.
Observer / Publish-Subscribe	Capas o módulos se comunican mediante eventos.	EventDispatcher implementa el patrón; listeners manejan eventos.	Patrón base en Svelte: stores y eventos reactivos.	Limitado a nivel de aplicación.	Nativo: WS usa subscripciones y emisión de eventos.
Mediator	Coordina interacción entre capas sin acoplamiento directo.	Mediadores entre controladores, servicios y eventos.	Coordinadores que gestionan comunicación entre componentes.	Mediador entre consultas y módulos.	Mediador entre múltiples conexiones WS.

Análisis de Mercado Laboral

Tecnología	Demanda Laboral	Fortalezas Mercado	Desafíos/Debilidades	Comentario Contextual
MVC + Capas	Alta demanda desarrolladores backend	Empresas buscan sistemas organizados	Exige conocimiento frameworks modernos	Fundamento valorado empresas tradicionales
Symfony PHP	Muy alta demanda Colombia	Buen salario senior/remoto	Competencia fuerte junior exige portafolio	Uno frameworks PHP más requeridos Colombia
SvelteJS	Demanda moderada startups frontend	Ventaja competitiva pocos dominan	Menor ofertas vs React/Vue	JavaScript moderno donde Svelte opción
SQLite	Poca demanda específica	Apps móviles proyectos pequeños	Rara vez mencionado explícitamente	Herramienta soporte más que rol principal
WebSocket	Demanda creciente tiempo real	Empresas valoran experiencia sockets	Requiere conocimiento especializado	Plus diferencial ofertas frontend/backend

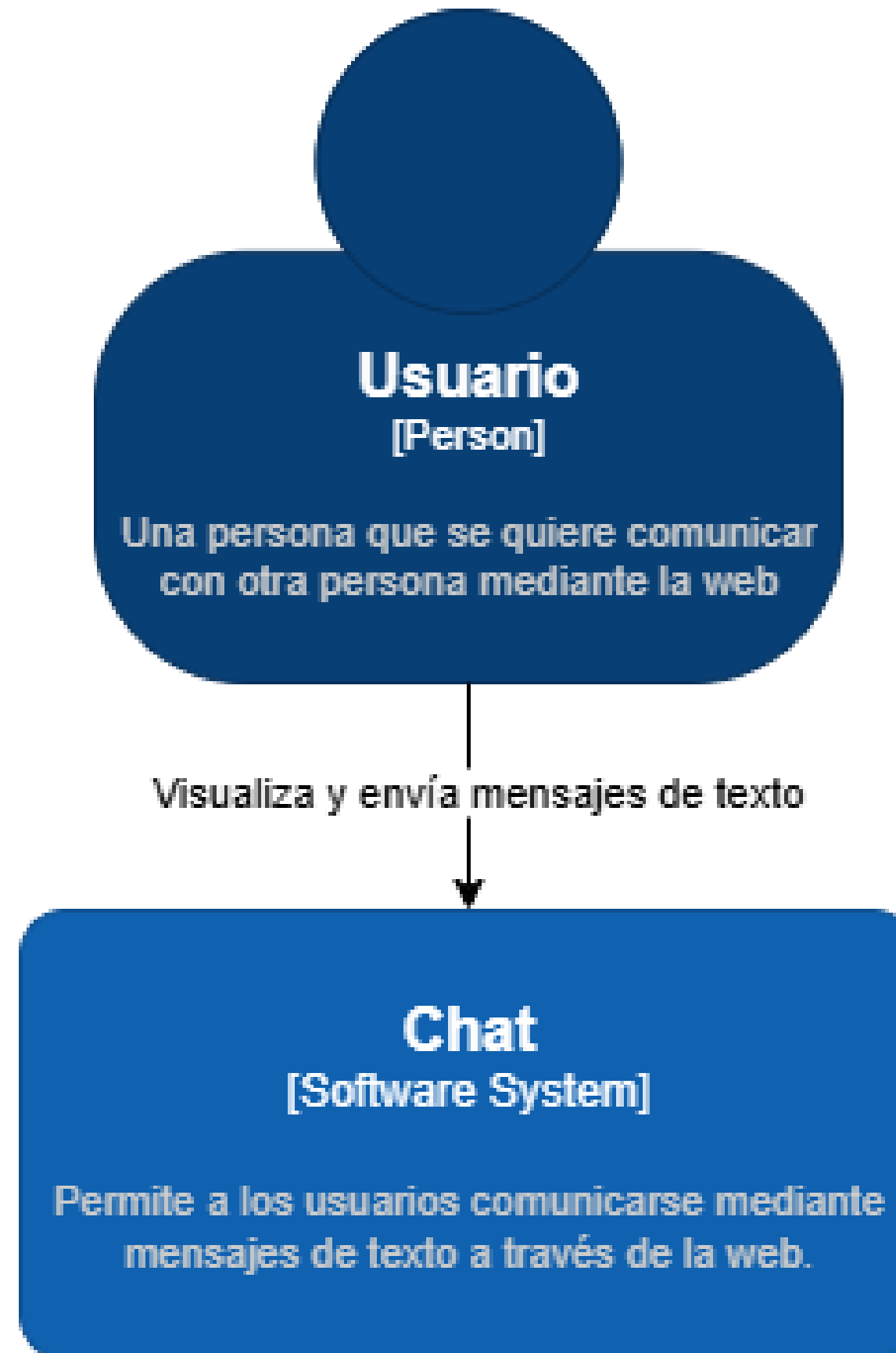
Ejemplo Práctico

Diagrama de Alto Nivel



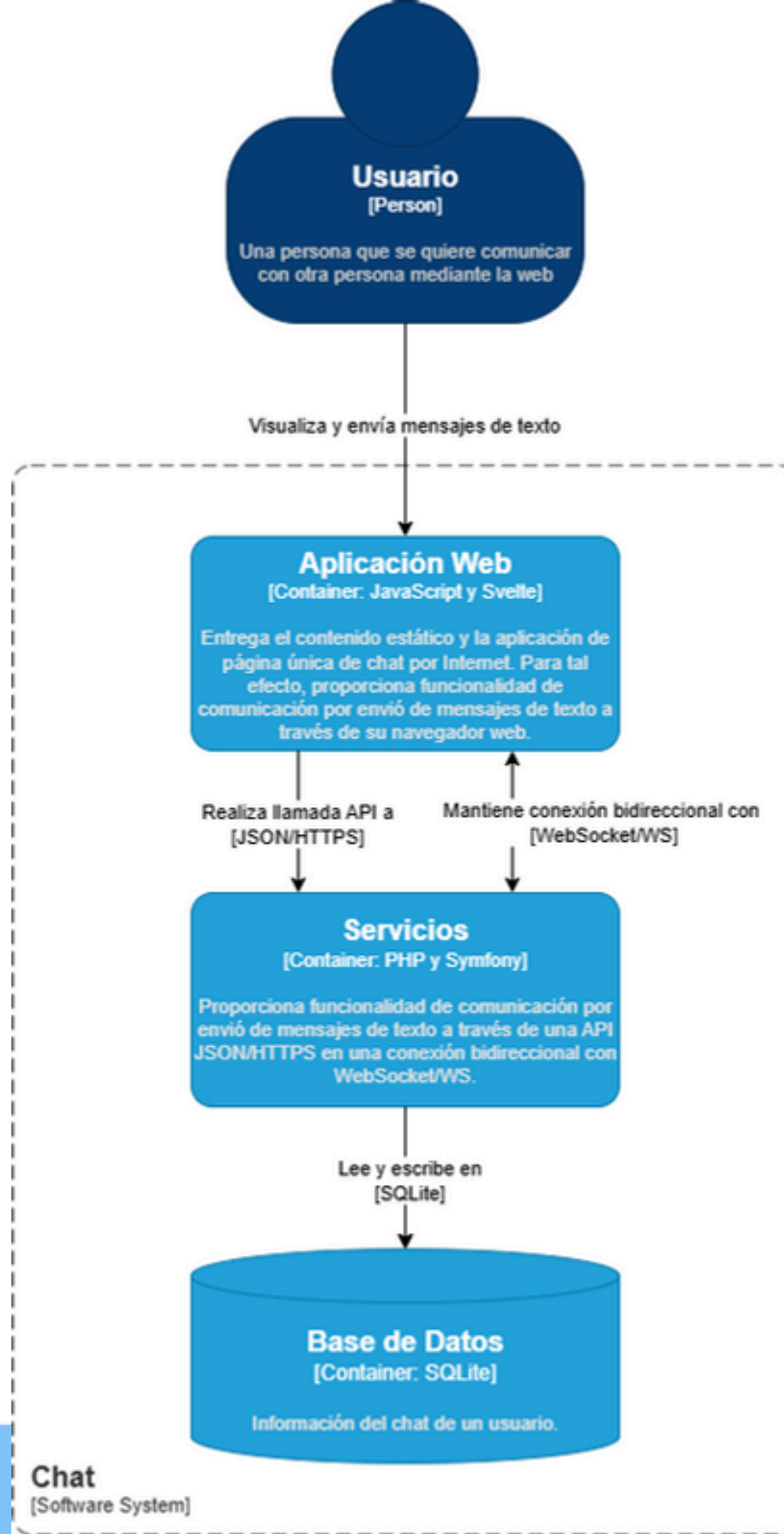
Ejemplo Práctico

Diagrama de Contexto



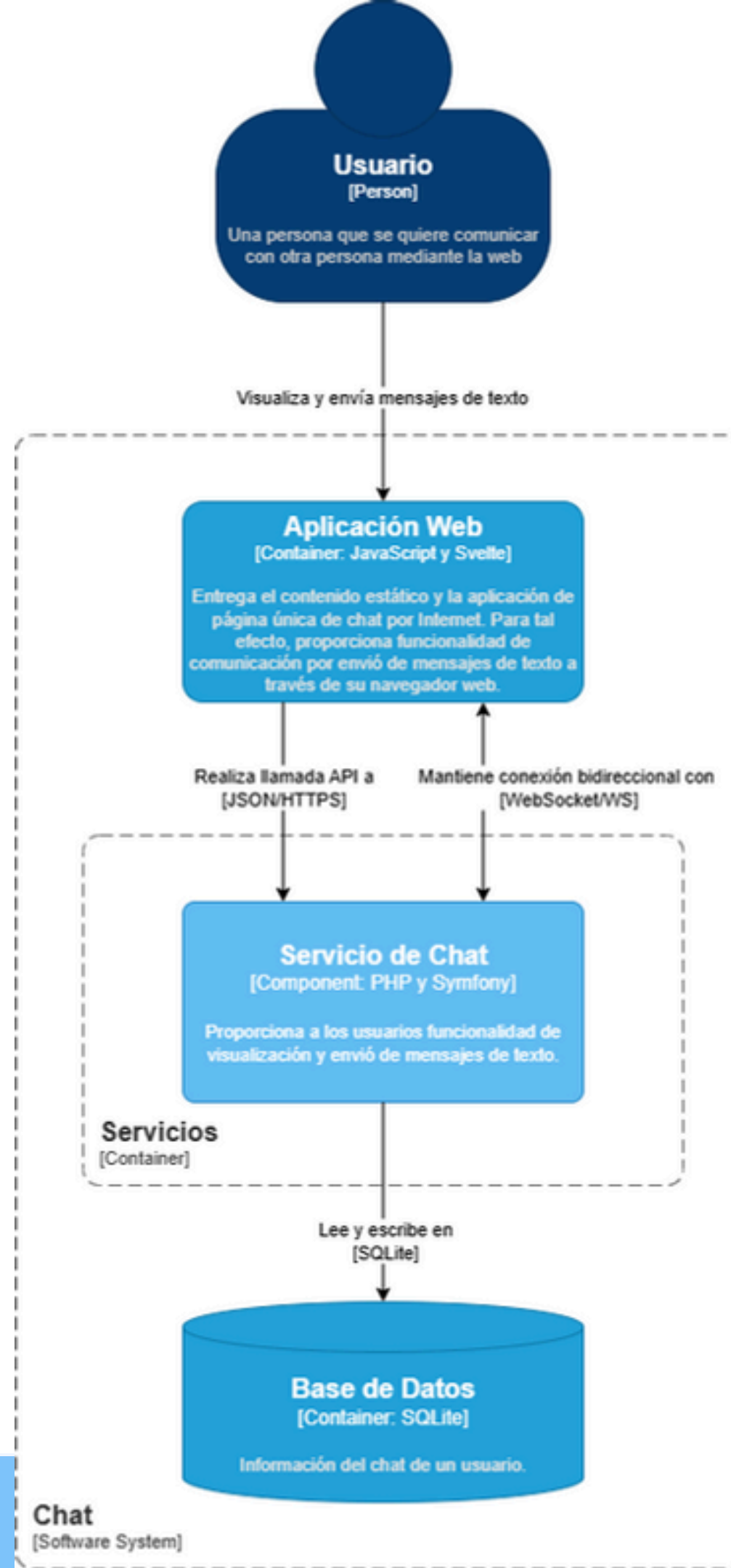
Ejemplo Práctico

Diagrama de Contenedores



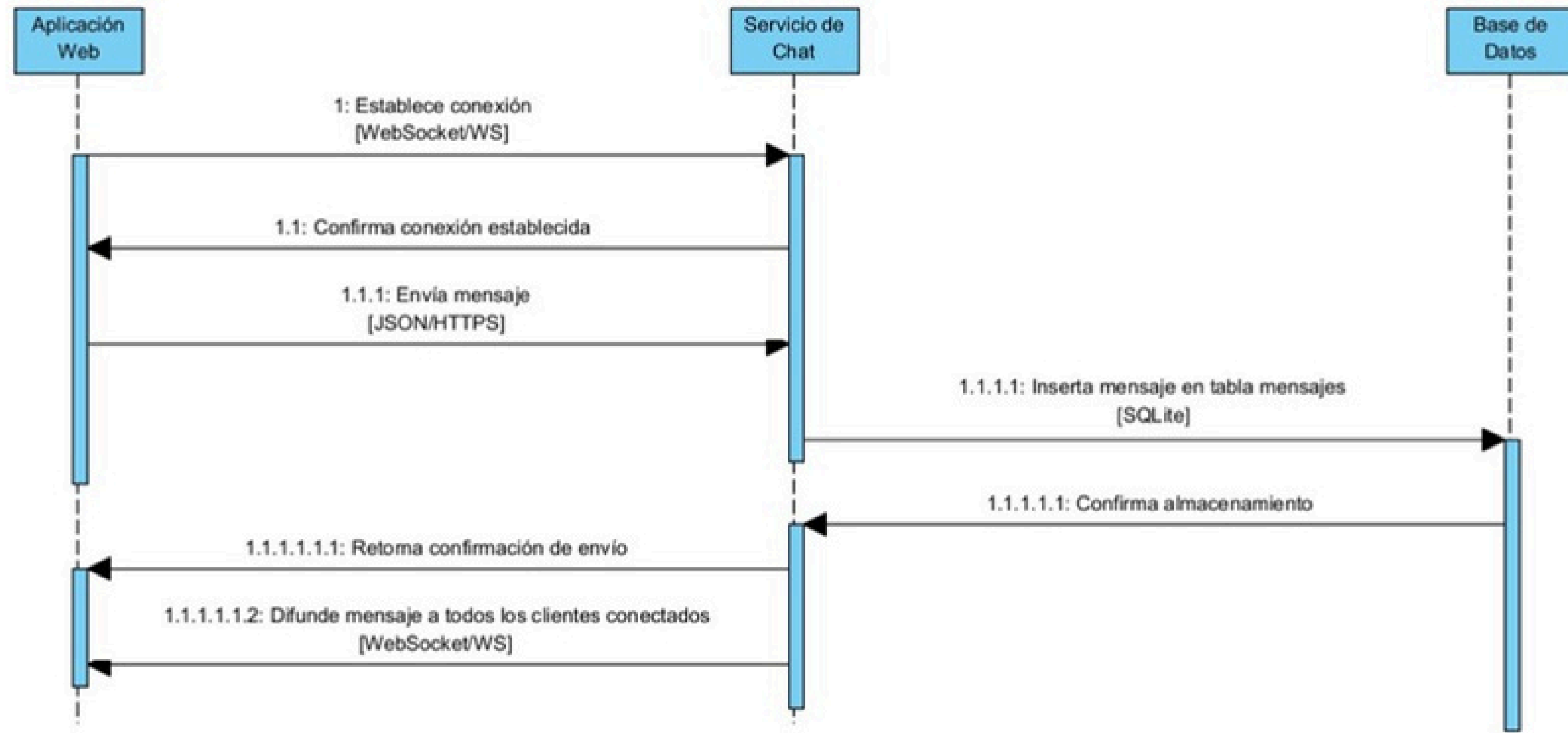
Ejemplo Práctico

Diagrama de Componentes



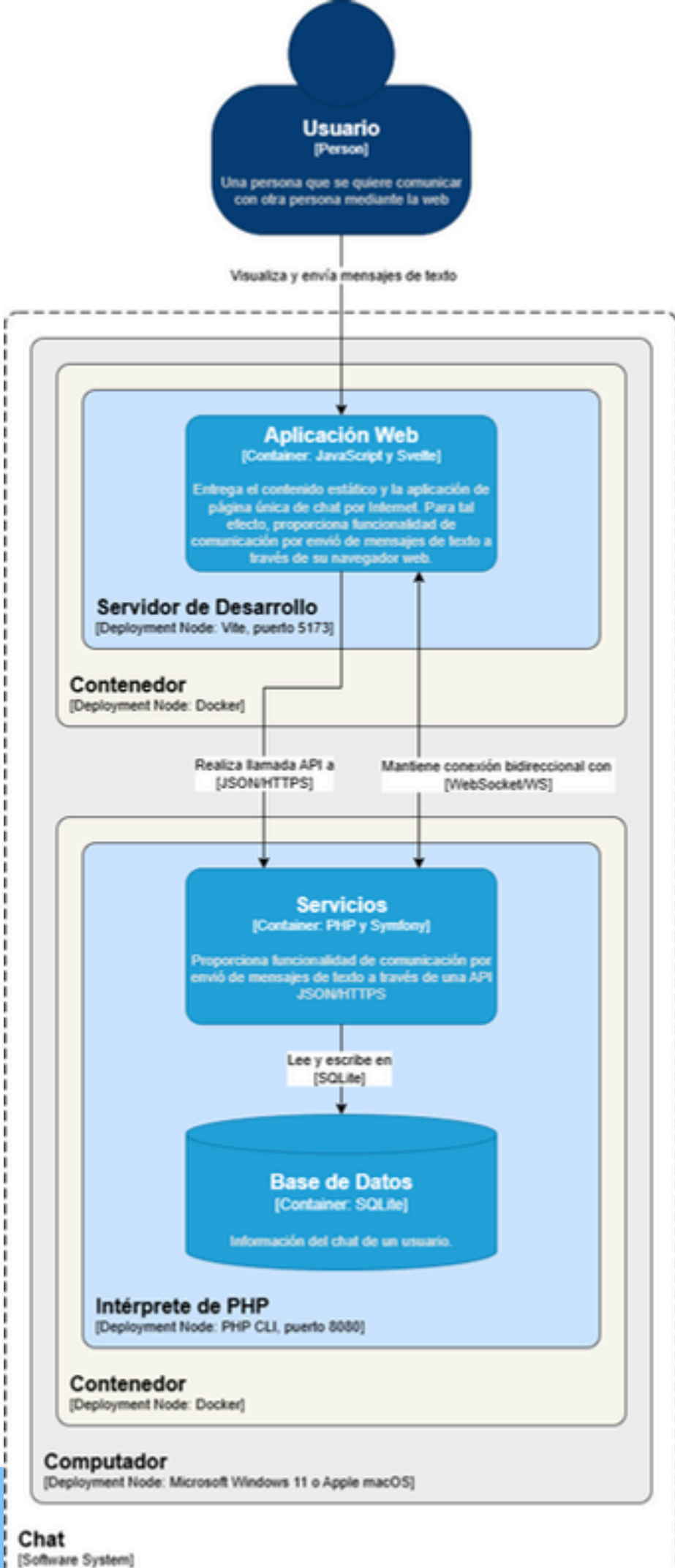
Ejemplo Práctico

Diagrama Dinámico



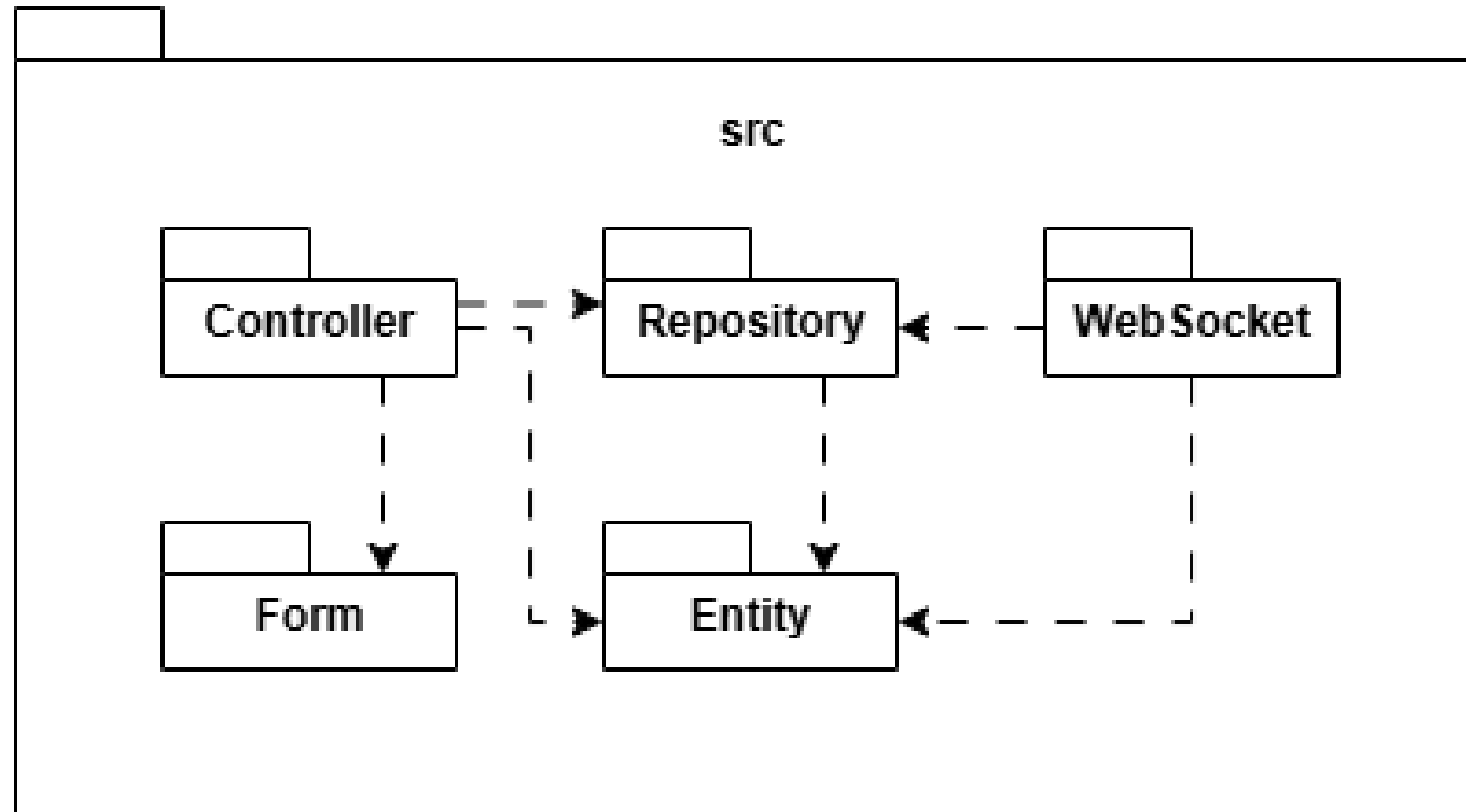
Ejemplo Práctico

Diagrama de Despliegue



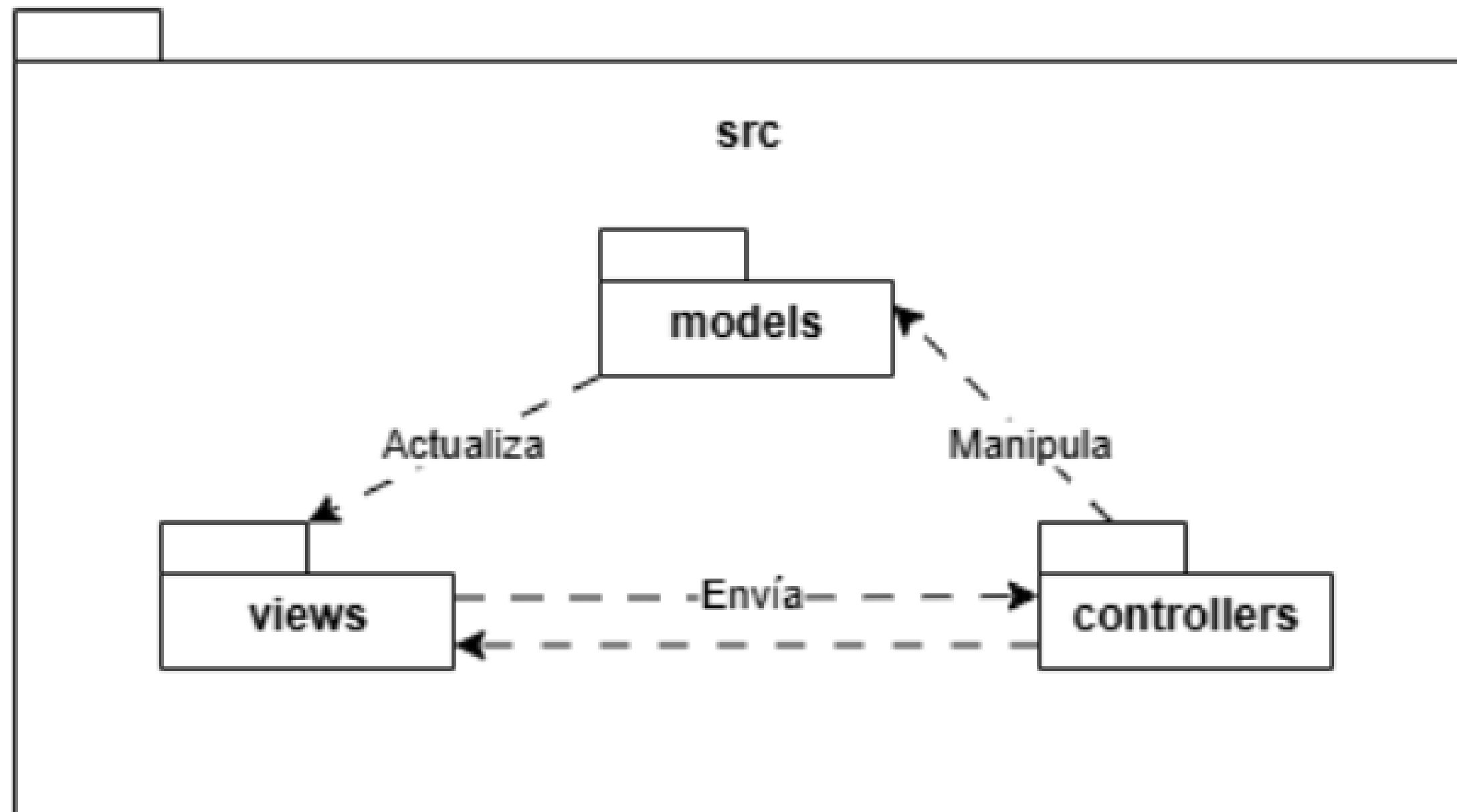
Ejemplo Práctico

Diagrama de Paquetes UML del componente Servicio de Chat



Ejemplo Práctico

Diagrama de Paquetes UML del componente dentro de Aplicación Web





Ejemplo Práctico

Repositorio GitHub

<https://github.com/adrianrrruiz/chat-app>

<https://github.com/adrianrrruiz/chat-app/releases/tag/FINISHED>

 `chat-backend` `chat-frontend` `docker` `docker-compose.yml`



¡Gracias!

