INFORME DE PROYECTO PARCIAL 2

SISTEMA DE GESTIÓN DE INVENTARIO Y ÓRDENES DE COMPRA

Universidad de las Fuerzas Armadas ESPE

Arico Cesar, Herrera Alan, Suquillo Fernando

FECHA: 17 DE NOVIEMBRE DE 2025

SANGOLQUI - PICHINCHA

Contenido

1. Descripción general del sistema

El Sistema de Gestión de Inventario y Órdenes de Compra tiene como objetivo apoyar a una empresa dedicada a la importación y distribución nacional de productos, permitiendo administrar de forma integrada el catálogo de productos, los proveedores, el inventario disponible en múltiples bodegas y el ciclo completo de las órdenes de compra.

El sistema facilita la creación, aprobación y recepción de órdenes de compra, así como la actualización automática del stock tras la recepción de mercadería. Adicionalmente, permite generar reportes básicos de abastecimiento, tales como productos críticos, historial de compras y gastos por proveedor.

Para el desarrollo de la solución se adopta una arquitectura de microservicios, lo que permite que cada componente del negocio pueda evolucionar, desplegarse y escalar de manera independiente, garantizando mayor flexibilidad, mantenibilidad y robustez del sistema.

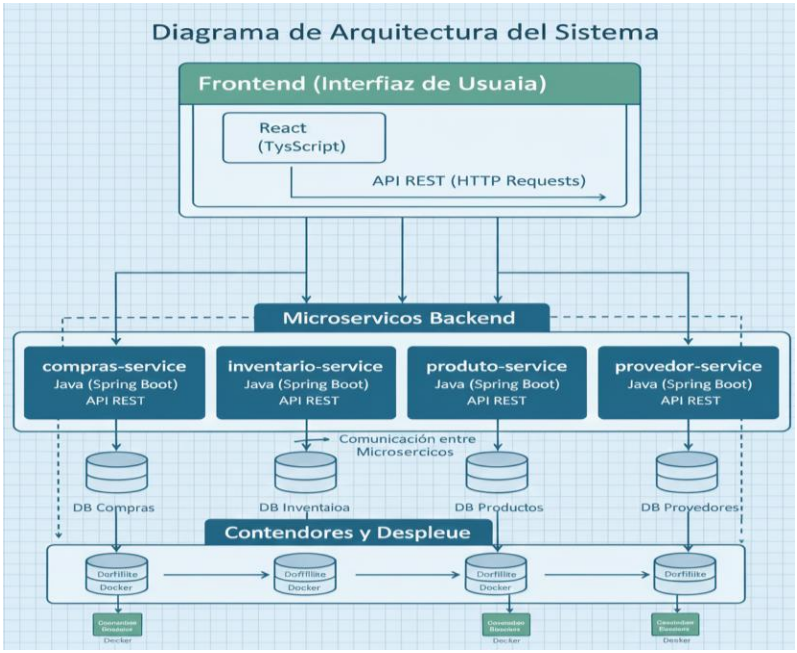
2. Arquitectura general del sistema

La arquitectura del sistema se basa en microservicios independientes desarrollados con Java 17 y Spring Boot 3. Cada microservicio posee su propia base de datos, siguiendo el patrón database-per-service, y se comunica con otros servicios mediante APIs RESTful utilizando HTTP y JSON.

El sistema se ejecuta en un entorno contenerizado mediante Docker, lo que asegura consistencia entre entornos de desarrollo y despliegue.

Figura 1

Estructura del Sistema de Microservicios y Frontend



Nota. Diagrama que representa la interacción entre el frontend en React y los microservicios de compras, inventario, productos y proveedores desarrollados en Spring Boot. Cada servicio cuenta con su propia base de datos independiente.

2.1 Componentes principales

- **Frontend (Interfaz de Usuario)**
  - Desarrollado en React con TypeScript.
  - Permite a los usuarios interactuar con el sistema para gestionar inventario, productos, proveedores y órdenes de compra.
  - Se comunica con los microservicios a través de APIs REST.
- **Microservicios Backend**

Cada dominio funcional está implementado como un microservicio independiente usando Java y Spring Boot.

  - Microservicios principales:
  - compras-service: Gestión de órdenes de compra.
  - inventario-service: Administración del inventario.
  - producto-service: Gestión de productos.
  - proveedor-service: Gestión de proveedores.

Cada microservicio expone su propia API REST y administra su propia base de datos.

- **Persistencia de Datos**
  - Cada microservicio tiene su propia base de datos, asegurando independencia y encapsulamiento de datos.
- **Contenedores y Despliegue**
  - Todos los servicios y el frontend cuentan con Dockerfile para facilitar el despliegue en contenedores.

3. Descripción de los microservicios

3.1 Servicio de Productos

- **Responsabilidad:**
  El servicio de productos es responsable de gestionar toda la información relacionada con los productos que forman parte del inventario de la empresa. Esto incluye la creación de nuevos productos, la actualización de sus características, la consulta de información detallada y la eliminación de productos que ya no se comercializan. Además, este servicio puede validar la existencia de productos y su información básica para otros módulos del sistema.
- **Entidades principales:**
  - **Producto:** Representa cada artículo gestionado en el inventario. Sus atributos principales incluyen:
    - id: Identificador único del producto
    - nombre: Nombre descriptivo
    - descripción: Detalles adicionales
    - precio: Valor unitario
    - stock mínimo: Cantidad mínima recomendada
    - categoría: Clasificación del producto
    - estado: Activo/inactivo

- **Endpoints REST clave:**
    - GET /productos — Permite obtener la lista completa de productos registrados en el sistema, con opción de filtros por nombre o categoría.
    - GET /productos/{id} — Devuelve la información detallada de un producto específico, útil para consultas o ediciones.
    - POST /productos — Permite registrar un nuevo producto en el sistema, validando que no exista duplicidad.
    - PUT /productos/{id} — Actualiza los datos de un producto existente, como precio, descripción o estado.
    - DELETE /productos/{id} — Elimina un producto del sistema, generalmente si ya no se comercializa o está obsoleto.

3.2 Servicio de Proveedores

- **Responsabilidad:**
  Este microservicio se encarga de la gestión integral de los proveedores, permitiendo registrar nuevos proveedores, actualizar su información, consultar detalles y eliminar registros. Es fundamental para mantener actualizada la base de datos de empresas o personas que suministran productos, facilitando la trazabilidad y la gestión de relaciones comerciales.
- **Entidades principales:**
    - **Proveedor:** Representa a cada entidad que abastece productos. Sus atributos principales incluyen:
        - id: Identificador único
        - nombre: Razón social o nombre comercial
        - contacto: Persona de contacto
        - dirección: Ubicación física
        - teléfono: Número de contacto
        - email: Correo electrónico
        - estado: Activo/inactivo
- **Endpoints REST clave:**
    - GET /proveedores — Lista todos los proveedores registrados, permitiendo filtros por nombre o estado.
    - GET /proveedores/{id} — Muestra la información detallada de un proveedor específico.
    - POST /proveedores — Permite registrar un nuevo proveedor, asegurando la integridad de los datos.
    - PUT /proveedores/{id} — Actualiza la información de un proveedor existente, como datos de contacto o estado.
    - DELETE /proveedores/{id} — Elimina un proveedor del sistema, generalmente si ya no se mantiene relación comercial.

3.3 Servicio de Inventario

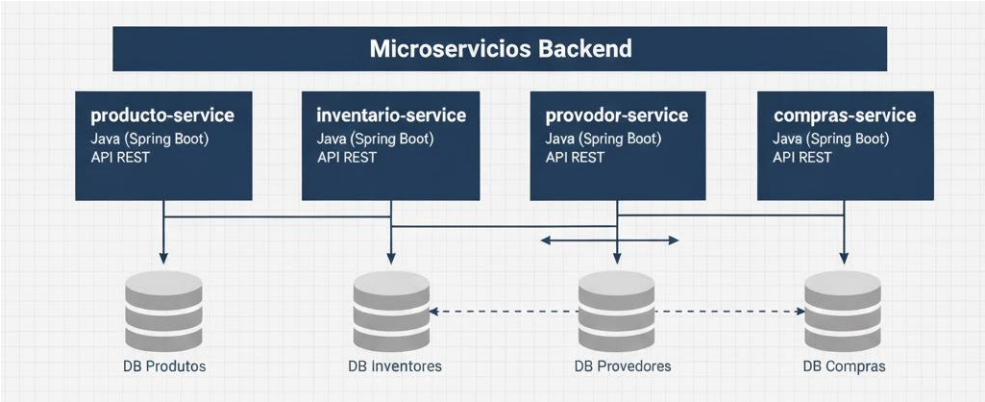
- **Responsabilidad:**
  El servicio de inventario es el encargado de controlar y mantener actualizadas las existencias de productos en los diferentes almacenes o ubicaciones de la empresa. Permite registrar movimientos de entrada y salida, consultar el stock disponible y asegurar que los niveles de inventario sean adecuados para la operación.
- **Entidades principales:**

    - **Inventario:** Representa el registro de existencias de cada producto. Sus atributos principales incluyen:
        - id: Identificador único
        - productoId: Referencia al producto
        - cantidadActual: Stock disponible
        - ubicación: Almacén o localización
        - fechaActualización: Última modificación
- **Endpoints REST clave:**
    - GET /inventario — Permite consultar el inventario completo, con opción de filtrar por producto o ubicación.
    - GET /inventario/{productoId} — Devuelve el stock actual de un producto específico.
    - PUT /inventario/{productoId} — Actualiza la cantidad disponible de un producto, por ejemplo tras una venta o reposición.
    - POST /inventario/movimiento — Registra un movimiento de inventario, ya sea entrada (compra) o salida (venta, merma).

3.4 Servicio de Órdenes de Compra

- **Responsabilidad:**
  Este microservicio gestiona todo el ciclo de vida de las órdenes de compra, desde su creación hasta su cierre. Permite registrar nuevas órdenes, consultar el historial, actualizar estados (pendiente, en proceso, completada, cancelada) y asociar productos y proveedores a cada orden. Es clave para la gestión de abastecimiento y control de compras.
- **Entidades principales:**
    - **OrdenCompra:** Representa cada solicitud de compra realizada a un proveedor. Sus atributos principales incluyen:
        - id: Identificador único
        - proveedorId: Referencia al proveedor
        - fecha: Fecha de emisión
        - estado: Estado actual (pendiente, completada, cancelada)
        - listaProductos: Detalle de productos y cantidades solicitadas
        - total: Monto total de la orden
- **Endpoints REST clave:**
    - GET /ordenes — Lista todas las órdenes de compra, con opción de filtrar por estado o proveedor.
    - GET /ordenes/{id} — Muestra los detalles completos de una orden específica, incluyendo productos y estado.
    - POST /ordenes — Permite crear una nueva orden de compra, asociando productos y proveedor.
    - PUT /ordenes/{id} — Actualiza el estado o información de una orden, como recepción de productos o cancelación.
    - DELETE /ordenes/{id} — Elimina una orden de compra, generalmente si fue creada por error o cancelada antes de su procesamiento

Figura 2
Estructura Detallada del Backend y Endpoints REST



Nota. Detalle de la capa lógica del sistema. Se visualiza la comunicación inter-servicios y la exposición de las API REST para la gestión de entidades mediante Java y Spring Boot.

4. Arquitectura del frontend

El frontend del sistema está desarrollado utilizando React con TypeScript, siguiendo una arquitectura modular y basada en componentes reutilizables. Esta estructura facilita el mantenimiento, la escalabilidad y la experiencia de usuario.

4.1 Estructura de componentes / módulos

El frontend se organiza en los siguientes módulos y componentes principales:

- **Componentes de Presentación:**

  Encargados de mostrar la información y la interfaz gráfica. Ejemplo: Layout, tablas de datos, formularios de entrada.

- **Páginas:**

  Cada página representa una vista principal del sistema, agrupando componentes según la funcionalidad.

  - DashboardPage: Vista general del sistema y métricas clave.
  - InventarioPage: Gestión y visualización del inventario.
  - ProductosPage: Administración de productos.
  - ProveedoresPage: Gestión de proveedores.
  - OrdenesCompraPage: Manejo de órdenes de compra.

- **Servicios:**

  Encapsulan la lógica de comunicación con los microservicios backend mediante llamadas HTTP (API REST).

  Ejemplo: productoService.ts, proveedorService.ts, inventarioService.ts, ordenCompraService.ts.

- **Tipos y Utilidades:**

  Definición de tipos TypeScript para las entidades principales y funciones utilitarias para manejo de datos.

4.2 Rutas principales

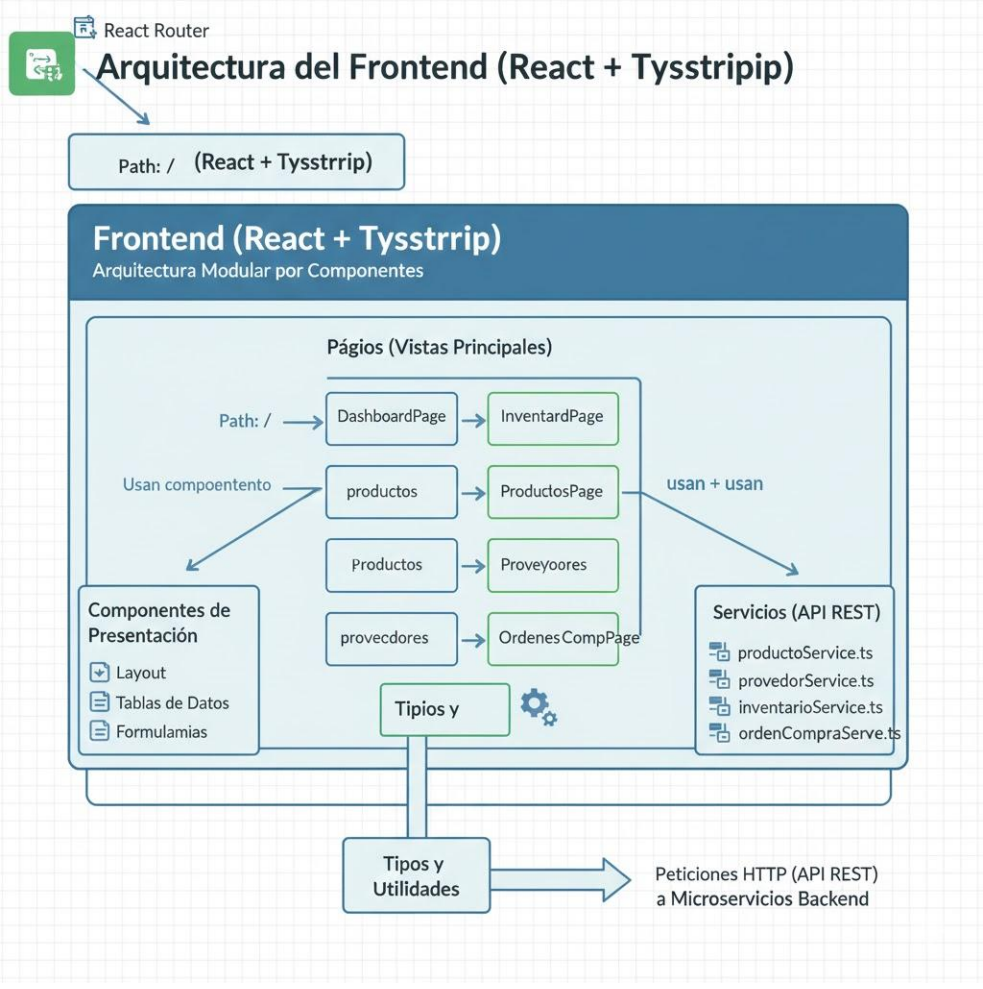
El sistema utiliza un enrutador (React Router) para gestionar la navegación entre las diferentes páginas. Las rutas principales son:

- / — Página principal o dashboard.
- /inventario — Gestión de inventario.
- /productos — Administración de productos.
- /proveedores — Gestión de proveedores.
- /ordenes-compra — Manejo de órdenes de compra.

Cada ruta está asociada a un componente de página, permitiendo una navegación fluida y una experiencia de usuario intuitiva.

Figura 3

Arquitectura Modular del Frontend y Enrutamiento



Nota. Diagrama de la estructura interna del frontend, resaltando la organización por componentes, servicios de consumo de API y el sistema de rutas (React Router) para la navegación del usuario.

5. Decisiones de diseño

5.1 Justificación del uso de microservicios

Se optó por una arquitectura de microservicios para lograr una mayor escalabilidad, mantenibilidad y flexibilidad. Esta aproximación permite que cada módulo del sistema evolucione de manera independiente, facilita la incorporación de nuevas tecnologías y mejora la tolerancia a fallos, ya que un error en un servicio no afecta a todo el sistema.

5.2 Razón para dividir el dominio

El dominio se dividió en servicios independientes (productos, proveedores, inventario y órdenes de compra) para reflejar las áreas funcionales clave del negocio. Esta separación permite asignar responsabilidades claras, reducir el acoplamiento entre módulos y facilitar el desarrollo paralelo por diferentes equipos.

5.3 Justificación de tecnologías

La selección de tecnologías para el desarrollo del sistema se realizó considerando criterios de robustez, escalabilidad, facilidad de mantenimiento, soporte de la comunidad y alineación con las mejores prácticas actuales en la industria del software.

- **Backend (Java + Spring Boot):**
  Se eligió Java como lenguaje principal por su madurez, rendimiento y amplia adopción en entornos empresariales. Spring Boot, por su parte, facilita la creación de microservicios gracias a su enfoque modular, su integración nativa con herramientas de seguridad, persistencia y pruebas, y su capacidad para simplificar la configuración y el despliegue. Además, Spring Boot cuenta con una gran comunidad y abundante documentación, lo que reduce la curva de aprendizaje y agiliza la resolución de problemas.
- **Frontend (React + TypeScript):**
  React fue seleccionado por su eficiencia en la construcción de interfaces de usuario dinámicas y reutilizables, así como por su ecosistema rico en librerías y herramientas. El uso de TypeScript añade tipado estático, lo que ayuda a prevenir errores en tiempo de desarrollo, mejora la mantenibilidad del código y facilita la colaboración en equipos grandes. Esta combinación permite construir aplicaciones web modernas, escalables y fáciles de evolucionar.
- **Contenerización (Docker):**
  Docker se adoptó para garantizar la portabilidad y consistencia del entorno de ejecución en todas las etapas del ciclo de vida del software. Permite empaquetar cada microservicio y el frontend con todas sus dependencias, asegurando que el sistema funcione de la misma manera en desarrollo, pruebas y producción. Además, facilita la integración con herramientas de orquestación como Docker Compose o Kubernetes, esenciales para sistemas distribuidos.
- **Comunicación (APIs RESTful):**
  Se optó por APIs RESTful para la comunicación entre el frontend y los microservicios, así como entre los propios servicios. Este enfoque promueve la interoperabilidad, la escalabilidad y la estandarización, permitiendo que otros sistemas o aplicaciones puedan integrarse fácilmente en el futuro.

6. Requisitos no funcionales implementados

El sistema implementa una serie de requisitos no funcionales que garantizan su calidad, usabilidad y robustez:

- **Validaciones exhaustivas:**

Todas las entidades y DTOs cuentan con validaciones para asegurar la integridad de los datos. Se **Manejo correcto de códigos HTTP:**
verifica la presencia de campos obligatorios, el uso de formatos válidos (por ejemplo, correos electrónicos y números de teléfono), y que los valores numéricos sean positivos. Esto previene errores y datos inconsistentes desde el backend.

- **Manejo correcto de códigos HTTP:**
  Las respuestas de la API utilizan los códigos HTTP adecuados para cada situación:
  - 200 (OK) para operaciones exitosas de consulta
  - 201 (Created) para creaciones exitosas
  - 400 (Bad Request) para errores de validación
  - 404 (Not Found) cuando un recurso no existe
  - 409 (Conflict) para conflictos de datos
  - Esto facilita la integración y el manejo de errores en el frontend.
  - Manejo centralizado de errores en backend:
  - Se utiliza @RestControllerAdvice en Spring Boot para capturar y gestionar de forma centralizada las excepciones, devolviendo mensajes claros y estructurados al frontend.
- **Manejo de errores claro en frontend:**
  El frontend muestra mensajes de error comprensibles y visibles para el usuario, permitiendo identificar y corregir problemas rápidamente. Se implementan alertas y feedback visual ante errores o acciones importantes.
- **Estructura modular y escalable:**
  El código está organizado por capas y/o por features, facilitando el mantenimiento, la escalabilidad y la incorporación de nuevas funcionalidades.
- **Experiencia de usuario (UI/UX):**
  La interfaz ofrece formularios claros, tablas legibles y feedback visual (loaders, alertas) para mejorar la interacción y la satisfacción del usuario.
- **Consumo eficiente de APIs REST:**
  El frontend consume los servicios backend mediante HTTP/JSON, gestionando rutas y estado de manera eficiente con hooks y contextos en React.

7. Contenerización y despliegue

Para facilitar la portabilidad, escalabilidad y despliegue del sistema, se implementó la contenerización de todos los microservicios y del frontend utilizando Docker. Cada componente cuenta con su propio archivo Dockerfile, lo que permite construir imágenes independientes y reproducibles.

- **Contenerización:**
  Cada microservicio backend (productos, proveedores, inventario, órdenes de compra) y el frontend tienen su Dockerfile, donde se definen los pasos para compilar, empaquetar y ejecutar la aplicación en un contenedor. Esto asegura que el entorno de ejecución sea consistente en desarrollo, pruebas y producción.
- **Despliegue:**
  Las imágenes Docker generadas pueden ser ejecutadas localmente o en cualquier plataforma compatible con contenedores, como servidores on-premise o servicios en la nube. Además, el sistema puede ser orquestado mediante herramientas como Docker

Compose o Kubernetes, permitiendo levantar todos los servicios y el frontend de manera coordinada y sencilla.

8. Mejoras futuras

Aunque el sistema cumple con los requisitos actuales, existen varias oportunidades de mejora y evolución para incrementar su funcionalidad, robustez y valor para el usuario:

- **Implementar autenticación y autorización:**
  Añadir un sistema de login y control de permisos para proteger los recursos y personalizar la experiencia según el rol del usuario.
- **Integración con sistemas externos:**
  Permitir la conexión con sistemas ERP, facturación electrónica o proveedores externos para automatizar procesos y mejorar la interoperabilidad.
- **Monitoreo y logging avanzado:**
  Incorporar herramientas de monitoreo (como Prometheus, Grafana) y logging centralizado para facilitar la detección y resolución de incidencias.
- **Automatización de pruebas y CI/CD:**
  Desarrollar pruebas automatizadas (unitarias, de integración y end-to-end) y pipelines de integración y despliegue continuo para mejorar la calidad y agilidad del desarrollo.
- **Optimización de la experiencia de usuario:**
  Mejorar la interfaz gráfica, añadir accesibilidad, soporte para dispositivos móviles y feedback visual más avanzado.
- **Escalabilidad y alta disponibilidad:**
  Implementar balanceo de carga, réplicas de servicios y estrategias de tolerancia a fallos para soportar mayor volumen de usuarios y garantizar la continuidad del servicio.
- **Notificaciones y alertas:**
  Agregar notificaciones por correo electrónico o en la aplicación para eventos importantes, como bajas de inventario o confirmación de órdenes.

9. Conclusión

El sistema de gestión de inventario desarrollado representa una solución moderna, escalable y robusta para la administración eficiente de productos, proveedores, inventario y órdenes de compra. La adopción de una arquitectura basada en microservicios, junto con tecnologías actuales como Java, Spring Boot, React y Docker, permite una alta flexibilidad, facilidad de mantenimiento y capacidad de evolución.

La implementación de buenas prácticas en validación, manejo de errores, experiencia de usuario y despliegue asegura que el sistema no solo cumpla con los requisitos funcionales, sino que también ofrezca una experiencia confiable y satisfactoria para los usuarios. Además, la estructura modular y la contenerización facilitan futuras mejoras, integración con otros sistemas y escalabilidad según las necesidades del negocio.

10. Referencias.

- Bass, L., Clements, P., & Kazman, R. (2021). Software Architecture in Practice (4th ed.). Addison-Wesley Professional.
- Richardson, C. (2018). Microservices Patterns: With examples in Java. Manning Publications.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
- Docker Inc. (2023). Docker Documentation. https://docs.docker.com/
- Pivotal Software, Inc. (2023). Spring Boot Reference Documentation. https://docs.spring.io/spring-boot/docs/current/reference/html/
- Meta. (2023). React – A JavaScript library for building user interfaces. https://react.dev/
- Microsoft. (2023). TypeScript Documentation. https://www.typescriptlang.org/docs/
- Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures (Doctoral dissertation, University of California, Irvine). https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm