STORAGIV

Infraestructura Virtual

José Alberto Gómez García

01

Estructura general

02

Implementación

03

Automatización

04

Tests

05

Contenedores

06

Integración continua

07

Servicios esenciales

08

Diseño de API REST

09

Implementación de API REST

Objetivos 0 y 1.
Estructura general y planificación.

Motivación del proyecto

- Reducir el número de errores y problemas que surgen a raíz de una mala organización del inventario y del registro de ventas en un negocio.
- Aportar datos de calidad que permitan analizar el mercado.
- La idea surge de mi propia experiencia en el negocio familiar.

Objetivos 0 y 1.
Estructura general y planificación.

Historias de usuario

- HU1. Como gestor, poder predecir con mínimo de 30 días de antelación, los productos de los que debo disponer en el almacén para suplir la demanda.
- HU2. Como gestor, quiero poder disponer de una estimación de la liquidez del negocio, en función de ingresos percibidos y a percibir, así como de gastos realizados y a realizar.
- HU3. Como gestor, quiero poder predecir la afluencia de clientes al local en determinadas dechas y franjas horarias.
- HU4. Como gestor, quiero poder disponer de un informe de los productos más vendidos en función del tipo de cliente.

Objetivos 0 y 1.
Estructura general y planificación.

Milestones

- M1. Sistema de gestión de almacén. (HU1)
- M2. Sistema de gestión de ventas. (HU2)
- M3. Sistema de informes. (HU3 y HU4)
- M4. Integración de servicios auxiliares.
- M5. API REST.
- M6. Despliegue en la nube.

Productos mínimamente viable que van evolucionando.

Objetivos 0 y 1.
Estructura general y planificación.

Lenguaje de programación. *TypeScript*

- Posee tipado fuerte y estático.
- Diferentes ámbitos de visibilidad, en particular, el privado.
- Soporta enumerados, clases abstractas e interfaces.
- Pensado para programación orientada a objetos y funcional.

Entorno de ejecución. *Node.js*

- Permite trabajar con *TypeScript* “out of the box”.
- Software maduro, con comunidad activa y usado por grandes empresas.
- Alternativas: *Deno*, entorno de ejecución multiplataforma que utiliza *TypeScript* por defecto.
- Sus principales desventajas son la necesidad de convertir los módulos de NPM y la falta de estandarización de ES Modules. Aun así, se pueden importar módulos por URL.
- Su principal ventaja es la ejecución de código en modo “sanbdox”.

Objetivo 2.
Comenzar la implementación.

- Aprender sintaxis del lenguaje y a manejar *git*.
- Creación de la clase “Producto”.
  Al ser la base del problema, relacionado con todas las historias de usuario.
- Familiarizarse con la transpilación a *JavaScript* y cómo ejecutar.
- Seguir estándares de documentación.
- Crear *issues* para representar problemas y organizar el trabajo a realizar.
  Asignar dichos *issues* a *milestones*.
  Referenciar a los *issues* en los *commit*.

Objetivo 3.
Automatizar operaciones.

Gestor de tareas. ***NPM***

- Es un gestor de paquetes, que tiene funciones básicas como gestor de tareas. Permite ejecutar scripts definidos en *package.json*
- Útil y sencillo para proyectos pequeños, como es nuestro caso.
- Evitamos añadir una nueva dependencia.

En objetivos posteriores, se automatizará construcción y publicación de contenedores *Docker* (mediane *Github Action)*, así como tests de CI.

Objetivo 4.
Framework de tests.

Se busca que cumplan los siguientes requisitos:

- Incluya biblioteca de aserciones.
- Cuanto más rápido, mejor.
- Informes del código cubierto.
- Ser selectivo con los tests a ejecutar.
- Buena documentación y fácil de usar/configurar.
- Compatibilidad con *TypeScript*.

Objetivo 4.
Framework de tests.

Se barajan dos opciones, *Jest* y *Mocha*.

Tras analizar ambas, se concluye que, aparentemente, son tremendamente similares.
Se elige *Jest* por los siguientes motivos:

- No necesita de otras librerías. No añadimos dependencias.
- Incluye biblioteca de aserciones. *Mocha* requiere de *Chai*.
- Sintaxis más concisa, aunque hagan falta distintas sentencias.
- Informes de cobertura de código.

Para proyectos de mayor tamaño, puede ser preferible utilizar *Mocha* debido a su mayor flexibilidad.
Jest está pensado principalmente para tests unitarios.

Objetivo 5.
Contenedores para tests.

Existen alternativas a *Docker*, pero ni llegué a planteármelas.

- No existe una imagen “oficial” de *TypeScript* en *DockerHub*.
- Hay alguna imagen base, pero de versiones antiguas y sin mantenimiento.
- Las versiones actualizadas incluyen dependencias que no necesito.

Partiremos de la imagen *node:16.13-alpine*

- Se elige 16.13 por ser una versión LTS con soporte hasta el 30 de abril de 2024.
- Versión *alpine* por ocupar espacio mínimo y no incluir apenas librerías.
  Instalaremos lo estrictamente necesario para nuestro proyecto.
  Evitamos incompatibilidades y mejoramos seguridad.

Objetivo 5.
Contenedores para tests.

Se crean dos *Github Action* para automatizar acciones.

- Reconstruir contenedor y publicarlo en *DockerHub* si varía el fichero de dependencias o el *Dockerfile*.
- Actualizar README en *DockerHub* si sufre algún cambio el archivo README.md del repositorio. Se utiliza parte de acción publicada por *@meeDemain*.

Personalmente, ha sido la parte que más me ha costado entender.
Principalmente, por el manejo del sistema de archivos del contenedor, permisos y usuarios.

Objetivo 6.
Integración continua.

Se deben configurar dos sistemas de integración continua.
Se busca que cumplan los siguientes requisitos:

- Configuración e integración en nuestro proyecto sencillas.
- Tener un plan gratuito.
- Permitir testear varias versiones del lenguaje fácilmente.
- Se proporcione buena retroalimentación de los posibles errores.

Adicionalmente, sería conveniente:

- Que los tests se ejecuten en paralelo.
- La interfaz de las webs sean prácticas y permitan realizar acciones como modificar ficheros, manejar ramas, etc.

Objetivo 6.
Integración continua.

Travis CI

- Plan de prueba de un mes para proyectos open-source, aunque para ello tengas que introducir la tarjeta de crédito. El plan para estudiantes es solo para *private builds*.
- Es el sistema de integración continua que más fácil permite testear varias versiones del lenguaje de programación. Se realizan los tests a partir del código fuente por este motivo.

Circle CI

- Gran número de minutos mensuales gratuitos para proyectos open-source, paralelismo e incluye soporte para contenedores *Docker*. Se pueden utilizar con configuración mínima.
- Interfaz web intuitiva y con funcionalidades útiles, como avisar de errores en fichero de configuración, corregirlo y pushearlo en el momento, etc.
- Necesidad de activar *Github Checks API* manualmente.

Se valora *Semaphore CI*, pero el periodo de prueba gratuita es de 14 días.

Objetivo 7.
Servicios esenciales.

Se deben añadir servicios de *logging* y de configuración remota.

Para el servicio de *logging*, se busca lo siguiente:

- Compatibilidad con *TypeScript*.
- Permitir salida a ficheros, en formato JSON.
- Introducir el menor overhead posible.
- Que aporte información del lugar del código donde se registra actividad.
- Mantenimiento activo, actualizaciones y correcciones de errores frecuentes.
- Configuración lo más sencilla posible.
- Es deseable que también pueda mostrar información por terminal de forma “bonita”.

Objetivo 7.
Servicios esenciales.

Se barajaron varias opciones, con distintos resultados:

- ***TSLOG***:
  - No reconocía donde se llamaba al logger.
  - Problemas para mostrar mensajes por terminal (se pisaban con *Jest).*
  - Información escasa al exportar en JSON.

- ***TYPESCRIPT-LOGGIN:***
  - Difícil de configurar.
  - Sin mantenimiento ni actualizaciones desde hace más de un año.

- ***WINSTON:***
  - Problemas con los *@types* para *TypeScript*.
  - Más de un año sin actualizaciones.

- ***PINO:***
  - Cumple todos los requisitos y no hubo problemas ni en instalación ni configuración.
  - En particular, exportar como JSON, buena información y poco overhead.
  - Plugins adicionales para embellecer salida por terminal.

Objetivo 7.
Servicios esenciales.

Para configuración remota no tenemos tantas opciones.

- ***DOTENV***:
  - Sistema para cargar pares clave-valor.
  - Necesidad de un fichero .env
  - Asegurarnos de no publicar el fichero de configuración en repositorio online.
  - Hay alguna que otra alternativa, pero son proyectos poco conocidos y en muchas ocasiones, con nulo o escaso soporte/mantenimiento.

- ***ETCD3:***
  - Almacenamiento de pares clave-valor de forma distribuida.
  - Nos permite manejar de forma distribuida los parámetros de configuración.
  - No lo llegamos a utilizar, se deja preparado para el futuro.
  - La principal alternativa es *consul*. Relativamente similar a *etcd*.

Objetivos 8 y 9.
API REST.

Los principales criterios para la elección del framework son:

- Servir máximo número de peticiones por unidad de tiempo.
- Facilidad de instalación/configuración y trabajar de manera ágil.
- Buen soporte con *TypeScript*.
- Compatibilidad con sistema de logging elegido *(Pino)*.
- Que tenga soporte para tests sin tener que levantar el servidor.
- Software con mantenimiento activo.

Objetivo 8.
Diseño de API REST.

Nos centramos en cuatro opciones:

- ***KOA***:
  - Soporte para *TypeScript y Jest*.
  - Relativamente similar a *express*, rendimiento incluido.
  - Muy conocido.
  - Buena documentación.

- ***Ts.ED:***
  - Funciona sobre *koa* o *express*.
  - Escrito en *Type*Script.
  - Sistema basado en clases y decoradores.
  - Pensado para “iniciar un proyecto desde cero”.
  - Por lo dos puntos anteriores, me sentí bastante abrumado y me parecía bastante engorroso ponerlo en marcha. Decidí seguir buscando otras opciones.

Objetivo 8.
Diseño de API REST.

Nos centramos en cuatro opciones:

- ***NestJS***:
  - Construido sobre *express* y escrito en *TypeScript*.
  - Para crear proyecto se necesita CLI. Tendría que reorganizar todo.
  - Sistema basado en clases y decoradores.
  - Configuración inicial relativamente compleja.

- ***FASTIFY:***
  - Soporte para *TypeScript*. Actualizaciones y mantenimiento frecuente.
  - Se puede integrar fácilmente en un proyecto existente.
  - Gran control sobre controladores y parámetros de forma sencilla.
  - Capacidad de ejecutar tests sin levantar servidor. *(inject)*
  - Utiliza *Pino* como logger, el cual tenemos instalado.
  - Buena documentación, fácil de entender.
  - Según varias comparativas, framework más rápido de los cuatro.

Objetivo 9.
Implementación de API REST.

Verbo GET

URI	DESCRIPCION	URI	DESCRIPCION
/	Mensaje de bienvenida.	/bills	Todas las facturas.
/status	Comprobar estado.	/bills/:id	Factura concreta.
/products	Todos los productos del almacén.	/bills/:id/product/:idp	Producto concreto en factura concreta.
/products/:id	Producto concreto del almacén.	/bills/:id/total	Importe de una factura concreta.

Objetivo 9.
Implementación de API REST.

Verbo POST

URI	DESCRIPCION
/products	Creación/actualización de un producto en el almacén.
/bills	Creación/reset de una factura.
/bills/:id	Creación/actualización de un producto en una factura concreta.

Identificadores del recurso a crear/modificar en el cuerpo de la petición.

Objetivo 9.
Implementación de API REST.

Verbo PUT

URI	DESCRIPCION
/products/:id	Modificar datos de un producto en el almacén. (1)
/bills/:id/product/:idp	Modificar datos de un producto en una factura concreta. (2)

Normalmente, PUT para modificar y POST para crear, pero no tiene porque ser estrictamente así.

- (1). Permite la creación del producto si no existiera.
- (2). No permite creación del producto, pues podría requerir creación de la factura, y es un comportamiento no deseado.

Objetivo 9.
Implementación de API REST.

Verbo DELETE

URI	DESCRIPCION
/products/:id	Borrado de producto del almacén.
/bills/:id	Borrado de una factura entera.
/bills/:id/product/:idp	Borrado de un producto concreto en una factura concreta.

Objetivo 9.
Implementación de API REST.

Verbo PATCH

URI	DESCRIPCION
/products/:id	Modificar cantidad de un producto en el almacén.
/bills/:id/product/:idp	Modificar cantidad de un producto en una factura concreta.

Normalmente, PATCH se emplea para modificar parcialmente un recurso existente. No se usa para crear. Lo más normal, es que sólo se modifique la cantidad de un producto, tanto en almacén como en facturas.

Objetivo 9.
Implementación de API REST.

Schema de Fastify

- Permite imponer condiciones/validaciones a los datos:
  - Campos obligatorios en cuerpo de petición.
  - Tipo de dato, valores/longitudes mínimas y máximas.
  - Parametrizar los IDs en las URI.
  - Si no se cumplen, HTTP400 automático con mensaje descriptivo.

- BODY para cuerpo de peticiones POST, PUT y PATCH.
- PARAMS para parámetros en URI.
- El schema recibe objetos (desde el punto de vista de *JavaScript).*

Se registran creación, modificación y borrado de objetos, así como los HTTP404. Postman para realizar alguna comprobación mínima al levantar servidor.

“Parser” propio para las facturas.

- Necesario ya que *Object.fromEntries(map)* no es “recursivo”.
- Para cada factura en el mapa de facturas, obtenía el mapa de productos vacío.
- Se utiliza para formar JSON válido.

STORAGIV

Infraestructura Virtual

José Alberto Gómez García