Ejercicio 2: Nomenclatura

```
funcion f(number1, number2, number3){
        let sum = number1 + number2;
        let mult = sum * number3;
        let sin = Math.sin(mult);
        return sin;
}
```

Ó

```
funcion sin(number1, number2, number3){
        let sum = number1 + number2;
        let mult = sum * number3;
        return Math.sin(mult);
}
```

Ejercicio 5: Arquitectura

1. Implementar una arquitectura orientada a microservicios en donde se divida la aplicación en varios componentes individuales comunicados mediante eventos con un patrón de Observador, pensando en la escalabilidad del proyecto, Por mencionar solo algunos:

   - Servicio orientado al procesamiento de pagos, conexión con pasarelas .. etc.
   - Servicio para manejo de notificaciones.
   - Servicio de búsqueda general (productos, precios, etc.), con consultas implementadas de forma eficiente para permitir una búsqueda global minimizando tiempos de respuesta.
   - Servicio para el manejo (CRUD) de la data de productos y su relación con categorías

2. Por su cercanía a la Web, utilizar Node con un servidor de Express o utilizar Nest.js, que en resumen es Node + TypeScript, que obliga a implementar un fuerte tipado de datos. Otra solución interesante podría ser Loopback.

3. Almacén de datos no-SQL, dependiendo del negocio y cuánto se estima crecer podría pensarse en alguna solución de almacenamiento que ofrece AWS, o una base de datos orientada a grafos (Las estudié en mi trabajo de

licenciatura y son utilizadas por los grandes e-commerces del mundo) o una sencilla base de datos en Mongo.

Ejercicio 6: Política de nombres para variables.

A. De los nombre de variables y funciones en el código:

1. Utilizar nombres mnemotecnicos para cada variable declarada en scopes globales o locales y para cada función implementada.
2. Las variables o nombres de funciones cortos ,(máximo 20 caracteres) deberán utilizar una notación ***lowerCamelCase***.
3. No deben emplearse variables o nombres de funciones largos, pero en caso de hacerlo, deberán utilizar una sintaxis de ***snake_case***.
4. Al utilizar un esquema orientado a objetos, los métodos de carácter privado deben anteponer el carácter _ al inicio y ser nombrados con lowerCamelCase. Estos métodos deben definirse al final del archivo de código. Ejemplo: *private _getSquareRoot(x);*
5. Al definir objetos locales (interfaces, tipos, DTO's.. etc) para representar o modelar la data obtenida de, o enviada a, la bases de datos, las propiedades de estos objetos deberán llamarse igual que los atributos definidos en la entidad dentro de la base de datos. De esta forma se garantiza una consistencia entre propiedades y un fácil acceso sin estructuras intermedias.
6. Utilizar la metodología ***BEM*** para nombrar las clases en CSS

B. De los nombres de tablas, atributos, relaciones en bases de datos:

1. Todo nombre de una Tabla o entidad deberá iniciar con letra mayúscula e identificarse con una única palabra, no como combinación de varias.
2. Los nombres de relaciones resultantes entre tablas o entidades, deberán expresarse con ***snake_case***.
3. Los nombres de propiedades o atributos deberán expresarse con ***snake_case***.

C. De los commits en Git.

1. Añadir un patrón de commits a la metodología de ramas utilizada, que tenga como base a Conventional Commits (https://conventionalcommits.org/es/v1.0.0-beta.2)

   Ejemplo: Si cada tarea/issue de Jira, se trabaja en una rama separada, cuyo nombre de rama es el Id de la tarea, un commit para esa rama podría ser:

   T-504: fix: Accept terms button on register page doesn't work.

   Si la app del front y la api del backend se tienen en el mismo repo con el objetivo de dockerizar y hacer CI/CD de manera más fácil un commit podría ser:

   T-504: fix(front): Accept terms button on register page doesn't work.

   Dando contexto de esta forma a que se hizo un commit de tipo Fix en el front

2. Crear commits explicativos, y en inglés.