# **PUNTO 1: Marvel Challenge**

El desarrollo de esta prueba se hizo pensando en 3 capas:

* CAPA DE PRESENTACION (VISTAS): Esta capa está compuesta por todo el código HTML que se le presenta al usuario en el navegador. Esta capa esta compuesta por los siguientes archivos:
  + **Index.html**, En este archivo se cargan todos los módulos que extienden a angularJS y los módulos, controladores y servicios que se crearon para el desarrollo de la prueba. Además, los CSS usados.
  + **consultarPersonajes.html**, en esta vista es donde se consultan los personajes por el patrón digitado en el campo de texto. Además, en esta página en el costado derecho se listan los comics favoritos agregados aleatoriamente y los que ha agregado el usuario.
  + **detallarPersonaje.html**, en esta vista es en donde se muestran los comics relacionados al personaje que se seleccionó en la página anterior.
* CAPA CONTROLADORA (CONTROLLERS): Esta capa es aquella que se encarga de escuchar y responder todo evento que el usuario haga sobre la capa web. Esta capa esta compuesta por el archivo “controller.js”, donde se definen los siguientes controladores:
  + **consultarPersonajesController:** Este gestiona cualquier interacción del usuario con la pagina “consultarPersonajes.html”
  + **detallePersonajeController:** Este gestiona cualquier interacción del usuario con la pagina “detallarPersonaje.html”
  + **detalleComicController:** Este gestiona cualquier interacción del usuario con el modal (ventanita del detalle del comic) que se encuentran replicada en las páginas “detallarPersonaje.html” y “consultarPersonajes.html”
* CAPA DE SERVICIOS: Esta capa es la que expone toda la lógica del negocio a la capa controladora. Esta capa esta compuesta por el archivo “models.js”, donde se define el servicio “modeloComicsFavoritos” que expone las siguientes funciones correspondientes a las reglas del negocio:
  + **consultarPersonajes(patron):listaPersonajesReturn:** Consulta los personajes que tengan un nombre que inicie la cadena patrón.
  + **consultarComic(comic): comicReturn:** Consulta el comic con URI enviada por el atributo resourceURI del objeto JSON del servicio.
  + **consultarComicsByCharacter(personaje):listaComicsReturn:** Consulta todos los comics relacionados al id del personaje.
  + **guardarComicFavorito(comic):boolean:** Guarda en localStorage el comic recibido como parámetro.
  + **guardarTresComicsAleatorios(listaPersonajes):boolean:** De una lista de personajes escoge 3 comic aleatoriamente de cualquiera de ellos y los guarda en localStorage.
  + **existeComicEnFavoritos(comic):boolean:** verifica que un comic exista en la lista de camics favoritos almacenadas en localStorage
  + **deleteComicFromListFavourites(comic)void:** Elimina un comic de la lista de comic favoritos.
* CAPA SUPPORT (UTILIDADES Y CONSTANTES): Esta capa, aunque de capa no tiene nada, es una capa transversal a todas las anteriores capas para el uso de constantes y funciones genéricas o de utilidades. Esta capa esta compuesta por el archivo “utilidades.js”, donde se definen las siguientes constantes y funciones:
  + **urlPersonajes:** constante que define la url base para consumir los services rest de Marvel.
  + **getFechaSistema():fechaFormateada:** devuelve una fecha en un formato definido.
  + **addAutenticacionUrl(url):urlConCredenciales:** le agrega a una url del API de Marvel las credenciales de autenticación.
  + **getNumeroAleatorio(desde, hasta):numero:** devuelve un numero entre un numero recibido en el parámetro “desde” y el numero recibido en el parámetro hasta.
  + **existeComicInArrayOfComics(listaComics, comic):boolean:** valida si el comic recibido en el parámetro “comic” existe en la lista de comic recibido en el parámetro “**listaComics**”.

# **PUNTO 2: CODE REFACTORING**

## Punto 2.1: Las malas prácticas de programación que en su criterio son evidenciadas en el código.

Malas practica:

1. NO TENER UN CAPA MODELO: No tener clases propias del dominio o modelo de la aplicación que se armen con aquellos datos que se necesitan mostrar. Lo desarrolle así sin estas clases ya que el dominio de la aplicación son los mismos objetos JSON que nos expone el servicio. Y por agilidad en la entrega, y que no se especificó ningún requerimiento no funcional, el análisis al que llegué fue no realizar clases propias, sino usar la misma estructura que me expone el rest de Marvel. La evidencia es que no existen clases como Comic o Character propias de la app.
2. FALTA DE MODULARIDAD EN LA CAPA WEB: Al iniciar el desarrollo no me percaté de que dentro de la carpeta de los recursos estaban las imágenes que decía cómo debía quedar el diseño gráfico de la app, por tanto, había creado mi propio esquema de navegación muy diferente. Al adaptar toda la lógica de desarrollo al esquema que Uds. me dieron, por temor a no cumplir con la solución en los tiempos, replique componentes (si me quedaba tiempo lo corregía), por ejemplo, repliqué el componente de la lista de Comics favoritos en la página de consultar personajes y en la página de la lista de Comics de un personaje. De la misma manera hice con el modal que detalla el comic.
3. NO SEPARAR LOS CONTROLLERS Y SERVICES EN ARCHIVO JS DIFERENTES: Cuando las aplicación demandan muchas líneas de código es mas fácil hacer mantenimiento a la app, si se modulariza la app y los controladores y servicios de este módulo se trabajan cada uno en archivo .js separados. Por tal razón al ver que esta prueba no demanda muchas líneas de código, fue suficiente mantener toda la aplicación como un solo modulo. La evidencia de esto es que solo existe un solo modulo, con un solo archivo js para todos los controladores y un solo archivo js con un servicios para los servicios de Characters y Comics.
4. NO APLICAR PRUEBAS UNITARIAS: No se hizo por motivos de tiempo, pero es claro que las pruebas unitarias garantizan que las reglas de negocio se mantengan durante el evolutivo del desarrollo.
5. NO DECLARACION DE CONSTANTES CON MAYUSCULAS SOSTENIDA: la declaración de constantes se debe hacer con mayúscula sostenida. La evidencia es que la declaración de la constante “**urlPersonajes**” no esta con mayúscula sostenida.

## Punto 2.2: Cómo su refactorización supera las malas prácticas.

<<Entiendo que este punto se debe entregar posteriormente después de hacer la refactorización.>>

# **PUNTO 3: PREGUNTAS**

## ¿ En qué consiste el principio de responsabilidad única ? ¿Cual es su propósito?

Consiste en que las entidades debe hacer solo una única cosa, es decir, lo que les corresponde, no deben cumplir con responsabilidades de otras entidades. El propósito de este principio es conseguir el grado de granularidad deseada para que el mantenimiento o soporte de éste no impacte en gran medida.

## ¿Que características tiene según su opinión “buen” código o código limpio?

Código limpio no es más que cuando se vea un bloque de código, este hable por sí solo, sin tener la necesidad de debuguiarlo. Sin embargo, hay que dejar claro que, para entender una funcionalidad a nivel de código, ayudaría mucho conocer la base de la arquitectura en la que fue creado, con el fin de entender saber las capas a las que abría que intervenir. Pero más allá de lo anterior las característica que este debe cumplir como mínimo son las convenciones que el propio lenguaje te propone: el nombrado de variables, constantes, métodos y clases. Además de las recomendaciones que se dan como buena practica, como es evitar niveles altos de complejidad, método con mas de 20 líneas es candidato a dividirse, una clase no puede superar un máximo de 1000 líneas, entre otras. Sin embargo también hace parte de este concepto las propias convenciones y estructura de organización de los fuentes que adopta la organización para la que trabajas.