# **¿Qué es el testing?**

En el mundo actual las tecnologías con las que vivimos necesitan código, esto ha hecho que el desarrollo de software se haya vuelto industrial, donde contamos con procesos y herramientas, como el **testing**, que nos permiten entregar más rápido proyectos de calidad a nuestros clientes **reduciendo el riesgo de errores en producción**.



## **¿Por qué hacer testing?**

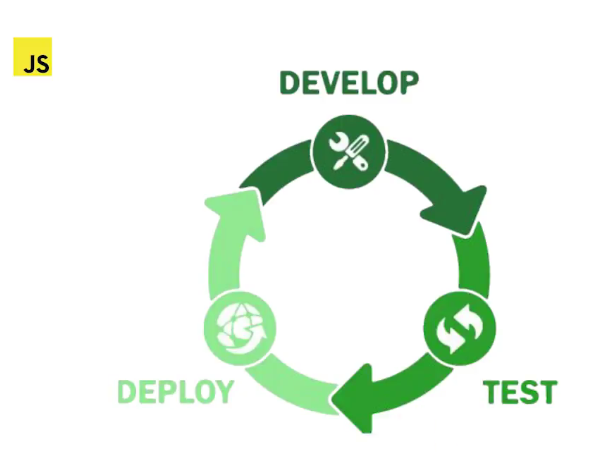
****Normalmente hay cuatro fases en el desarrollo de software para crear un producto:

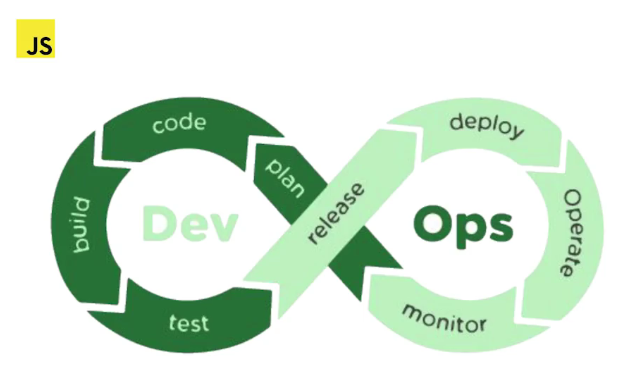
**Diseño ⇒ Desarrollo ⇒ Pruebas ⇒ Producción**

En estas fases, detectar un error se vuelve cada vez más costoso. Ahí es donde hacemos técnicas para prevenir estas situaciones. El **testing** es la manera en la que nosotros gestionamos el **riesgo** y tratamos de evitar, en la medida de lo posible, los errores en el sistema.

## **Gestionar riesgos como Google**

* **Análisis de código estático**: Mientras desarrollamos vamos viendo nuestros resultados.
* **Pruebas unitarias**: Nos aseguramos que el código funciona como queremos.
* **Pruebas de integración**: Verificar que varios elementos funcionan bien trabajando juntos.
* **Revisión de código**: Equipo o persona encargada de revisar el código de las demás personas.
* **QA**: Equipos de creación de pruebas automáticas o manuales.

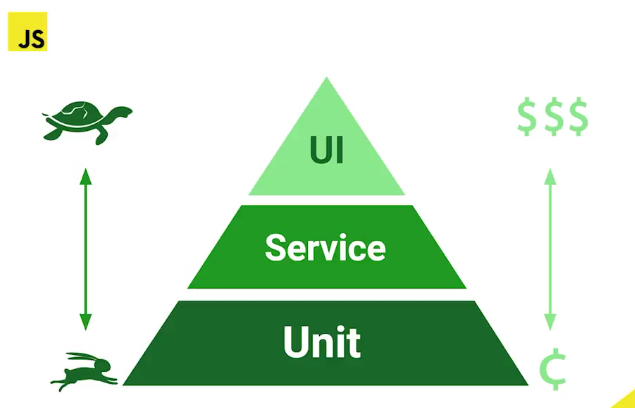


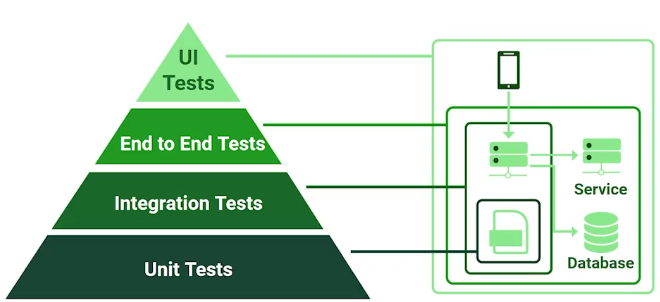


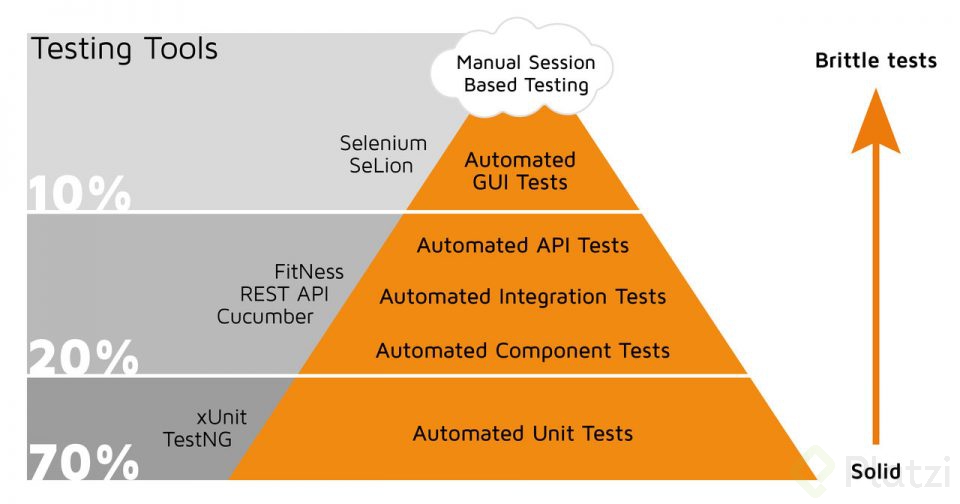
# **La pirámide del testing**

La pirámide clásica del testing (de **menos** a **más** costoso y tendría que haber más pruebas de las primeras que de las últimas)

* **Unit tests**.
* **Integration tests**.
* **End to end tests** (e2e)
* **UI test**: Emulan el dispositivo físico.

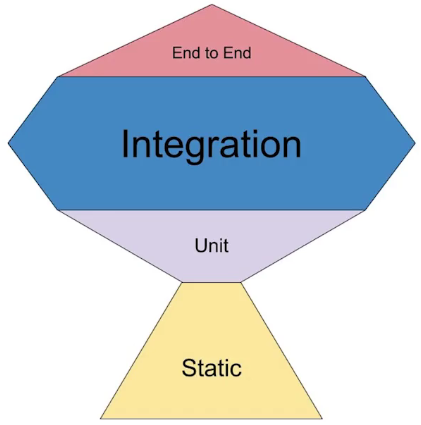






En javascript se recomienda en vez la pirámide el trofeo de tests:

* **Static tests**: el tronco (el segundo mayor número de pruebas)
* **Unit tests**: Mismo núm de pruebas que e2e y menos que static e integration.
* **Integration tests**: el mayor número de pruebas debe ser aquí.
* **e2e tests**: mismo tamaño que unit tests y menos que static e integration.



# **Deuda Técnica**

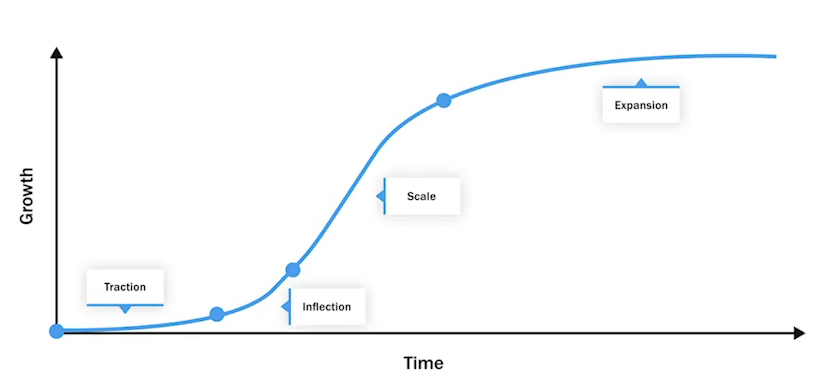
Igual que en las finanzas, las deudas no son buenas ni malas, solo son una estrategia para alcanzar algo y luego se paga.

## **Haciendo pruebas se maneja el “riesgo”**

* Diseño.
* Desarrollo.
* Pruebas.
* Producción.

El **momento** y **pruebas** dependen de la fase en la que se encuentra la compañía:

* Tracción.
* Inflexión.
* Escala.
* Expansión.



La **deuda de deficiencia** del desarrollador puede ser alta porque en la fase de **tracción**, la compañía busca **velocidad >** **precisión**, de modo que en esta etapa el testing no se valora, ya que buscan **lanzar, lanzar, lanzar...**

Cuando se entra a la fase de **inflexión**, hay más usuarios y se empiezan a escribir más pruebas, curando la deuda.

# **Herramientas de Testing**

Las herramientas mencionadas aquí están clasificadas según el tipo o nivel de pruebas a realizar.

## **Multipropósito o Robustas**

Pueden ejecutar la mayoría de pruebas de la pirámide o trofeo, tanto para backend, frontend, integration test, algunas hasta UI y API s testing.

* Jest
* Chai
* Mocha
* Ava
* Jasmine

## **UI Testing**

Pruebas desplegadas en un dispositivo real o simulado bajo un Browser (navegador) automatizando toques y gestos en la aplicación.

* Playwright
* Cypress
* WebdriverIO
* Puppeteer

## **API Testing**

En general, las APIs no tienen UI, usan protocolos de servicios por peticiones.

* Supertest
* Dredd
* Postman
* Insomnia

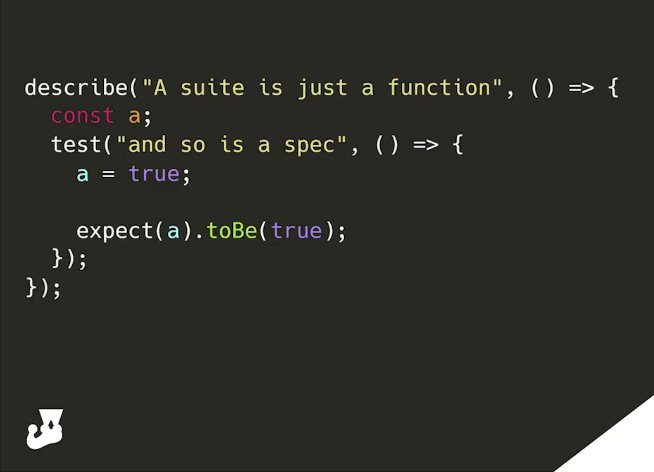
## **Pruebas Estáticas**

Estas pruebas no ejecutan el código, estas herramientas se integran al editor de código y van analizando si hay alguna mala práctica.

* ESLint
* Prettier
* Typescript tools (TSLint)

**Todas** estas herramientas manejan una sola estructura (de forma), de modo que el lenguaje o framework pasa a un segundo plano.

## **Test en Jest**



## **Matcher en Jest**

