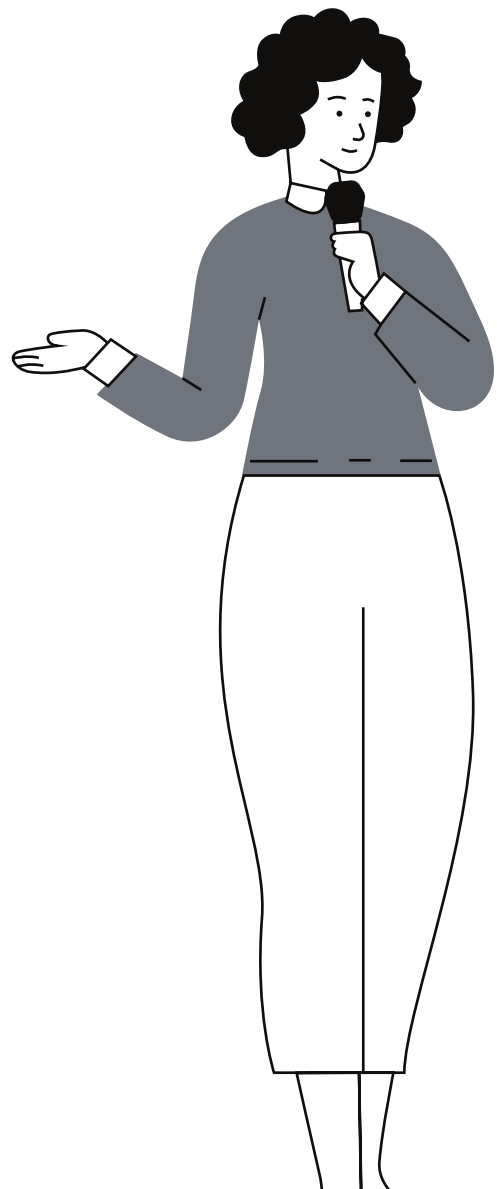


# Agenda



1

Web components, estándares y configuración del entorno de trabajo

2

Dominio de Stencil con JSX y Typescript a través de la creación de web components

3

Composición de componentes e interacción entre ellos

4

Testing, accesibilidad y otras herramientas

5

Publicación en npm de los componentes creados e integración con Angular

whoami

**Rafa Bernal**

Programador Frontend  
{pixel developer & Stencil Advocate}

@rbernalBer

**autentia**



### Un poco de historia

Los web components (a partir ahora **componentes** a secas) llevan años con nosotros. Estaban tan integrados entre nosotros que eran parte de nuestro día a día. Ej: select, video, audio... etiquetas html que hacían su función y nos preocupábamos de cómo lo hacían

```
<video src="video.mp4" width="640" height="480" autoplay muted loop></video>
```

Si quisiéramos hacer algo más elaborado, necesitábamos de la ayuda de JS. ¿**jQuery**? reconócelo, no ha pasado tanto tiempo desde la última vez que usaste:

```
$("#datepicker").datepicker();
```

### ¿Qué son?

Los **web components** es un **estandar** y son el fruto de un grupo de API's que nos permiten encapsular bloques de código (HTML, CSS y JS) que podemos reutilizar a lo largo de toda la web.

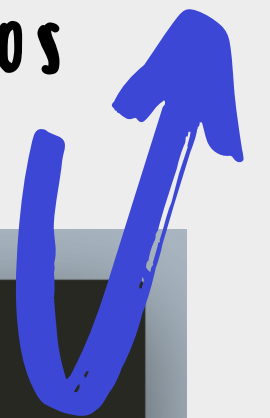
Estos bloques se comunican con el exterior a través de **atributos** y **eventos**

En resumen, nos permite crear **custom HTML elements** (etiquetas HTML personalizadas)

ATRIBUTOS



EVENTOS



```
<!DOCTYPE html>
<html>
  <head>
    ...
  </head>
  <body>
    <my-component first="Stencil" last="'Don't call me a framework' JS"></my-component>
  </body>
</html>
```

### El estandar

#### Custom Elements

La principal, nos permite crear nuestros propios elementos HTML personalizados. Con su comportamiento único y CSS propios.

#### HTML Templates

Es el mecanismo que nos permite encapsular el HTML y renderizarlo cuando lo solicitemos y no al cargar la página.






























#### Shadow DOM

El encargado de encapsular . Es capaz de mantener nuestros estilos limpios, ocultos y separados de otros componentes de nuestra app. El API de DOM Shadow es clave.

#### ES Modules

Nos permite importar/exportar código de diferentes archivos JS. Permitiendo la reutilización de nuestro código. Uno de los conceptos centrales detrás de los componentes web.

El estandar

Browser support	 CHROME	 OPERA	 SAFARI	 FIREFOX	 EDGE
 HTML TEMPLATES	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 CUSTOM ELEMENTS	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 SHADOW DOM	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE
 ES MODULES	 STABLE	 STABLE	 STABLE	 STABLE	 STABLE



¿Por qué es el momento?

La respuesta es simple, los **frameworks**, tan necesarios en nuestro día a día. Hasta la llegada de estos, los web components estaban pero no estaban.

Cada nuevo framework que aparecía venía con una gran idea bajo el brazo, ¿sabéis cual?; los componentes, que poco tenían que ver al principio con los web components actuales.

Cada nuevo framework traía un nuevo estándar. Y cada uno de ellos es un estándar competente.

```
import { Component } from "@angular/core";

@Component({
  selector: 'app-hello',
  styleUrls: [...],
  template: `
    <p> {{ greeting }} </p>`
})
export class AppHello {
  @Input() greeting:string = 'Hello'
}
```

```
import React from "react";
import "../styles.css";

function Hello({greeting}) {
  return (
    <p>{{ greeting }} World!</p>
  );
}

export default Hello;
```

```
<script>
  module.exports = {
    data: function () {
      return {
        greeting: 'Hello'
      }
    }
  }
</script>

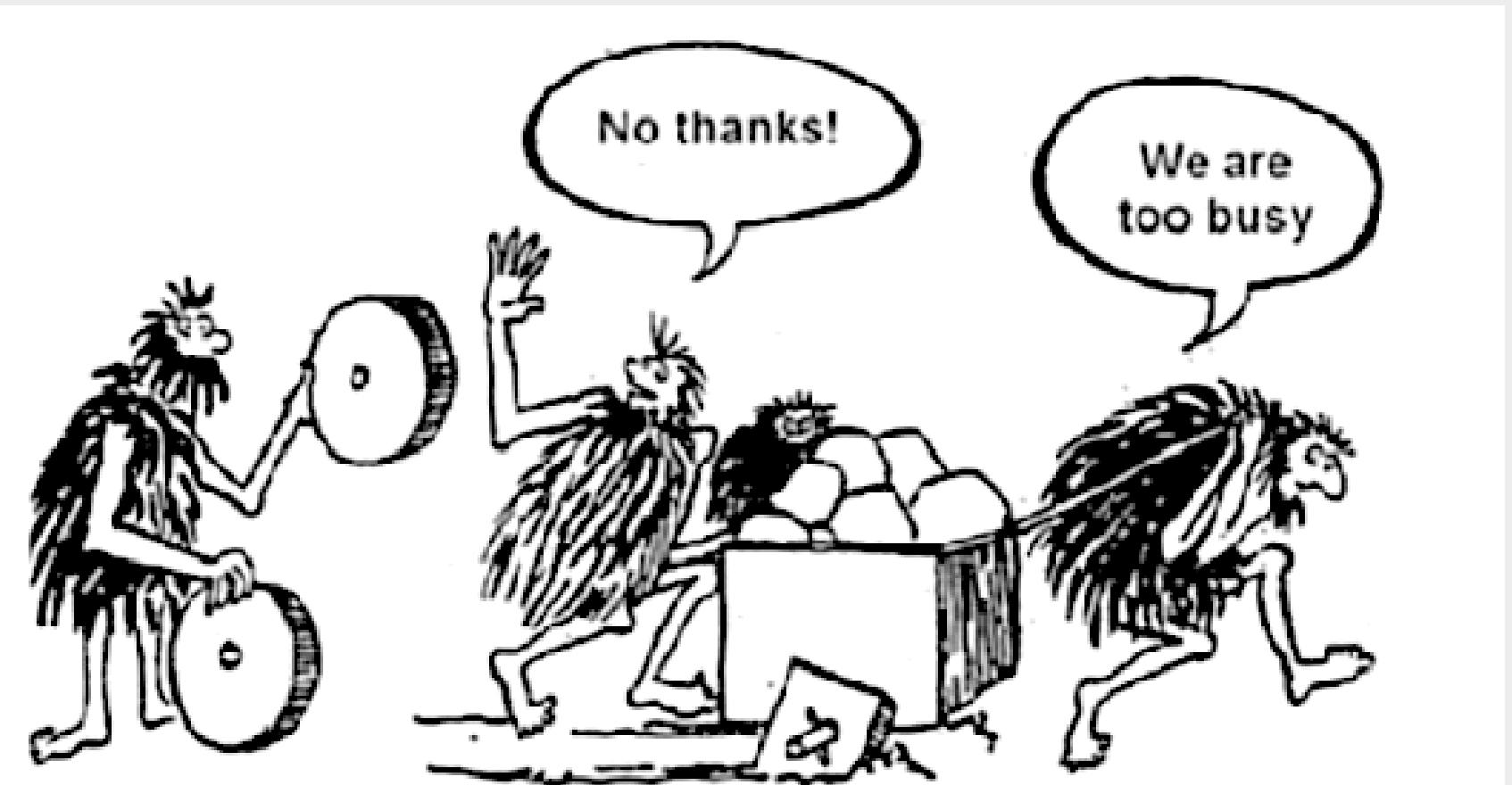
<style scoped>
  p {
    font-size: 2em;
    text-align: center;
  }
</style>
```



¿Por qué es el momento?

¿Alguien lleva la cuenta de cuántos front frameworks existen?  
EmberJs, Svelte, Preact, Alpinejs, Stimulus... y así una larga lista.

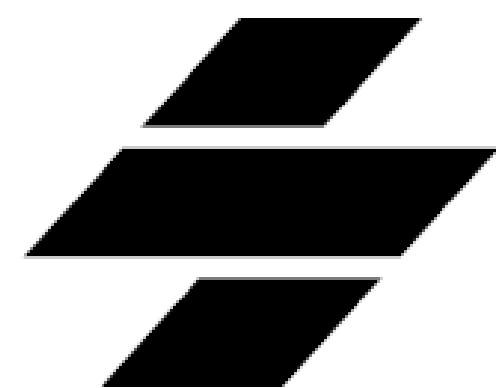
Cada uno con su gran idea bajo el brazo.  
**¡Revolucionarios!**



¿Qué papel juega aquí el estandar?



+info: [Custom Elements Everywhere](#)



# stencil

EL



BOSS!!

LA  
HERRAMIENTA  
MÁQ  
UINA

## ¿Qué es Stencil?

**Stencil** es una herramienta que nos permite construir componentes web, rápidos, 100% basados en los estándares y totalmente agnósticos.

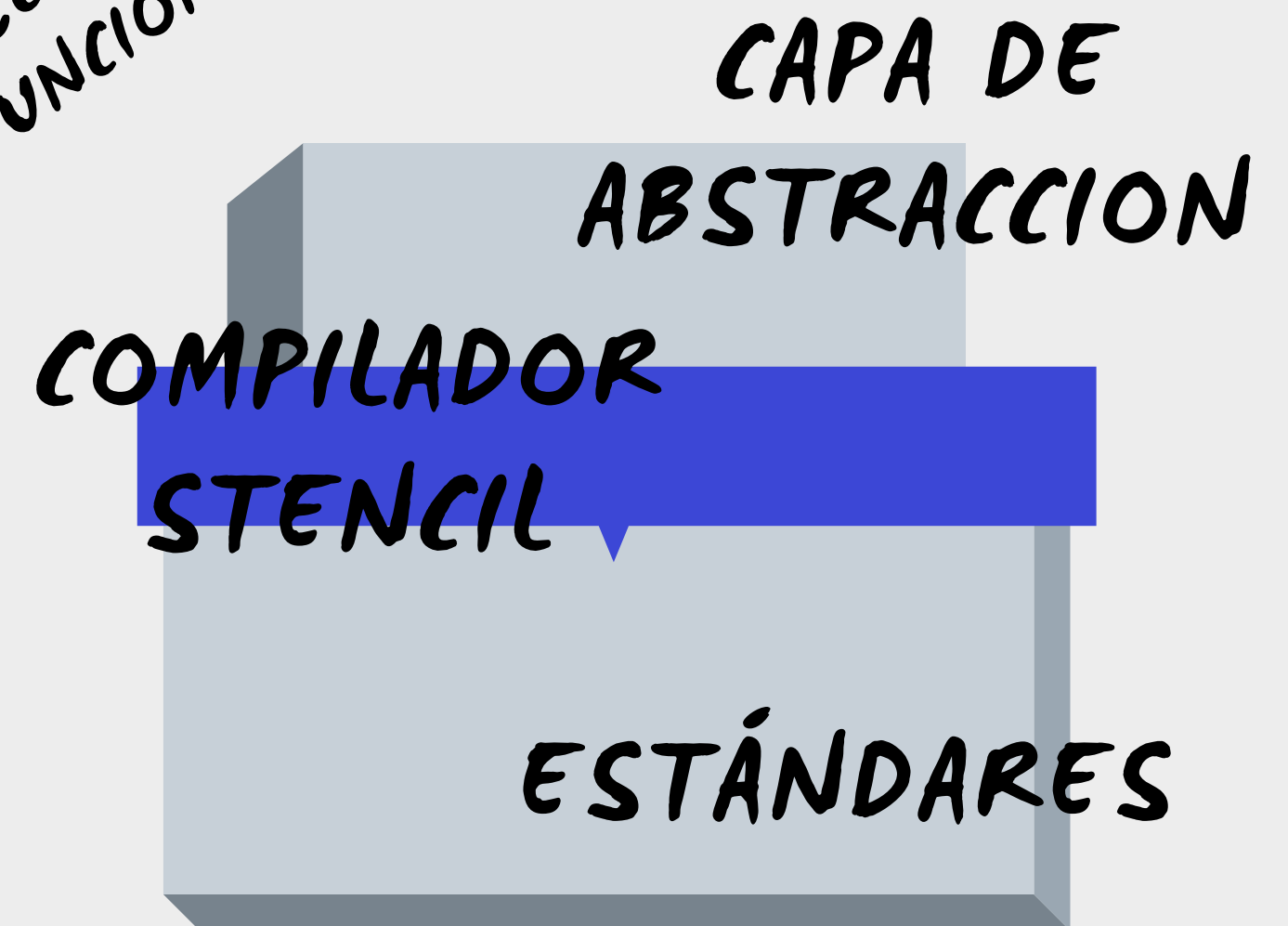
Lo creas una vez y lo usas en el framework de turno.

Comparándolo con la creación de componentes nativos, Stencil proporciona APIs adicionales que simplifican la escritura de componentes, mientras se mantiene el 100% de compatibilidad con la web

1

Web components, estándares y configuración del entorno de trabajo

¿CÓMO  
FUNCIONA?



¿Qué lo hace tan especial?

# ¡Stencil, no es un framework!

Stencil combina los mejores conceptos de los frameworks más populares en una sencilla herramienta de construcción.

**01** Virtual DOM

**02** Renderización asíncrona

**03** Reactive data-binding



**04** TypeScript

**05** JSX (...TSX)

CODE  
TIME!!

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

npm, es un gestor de paquetes para Node. Cuando instalamos Node en nuestro equipo, este a su vez instala npm. Esto no quiere decir que Node sólo pueda trabajar con NPM, también lo puede hacer con otros gestores como YARN.

En este curso trabajaremos con npm. Este gestor de paquetes contiene todos los paquetes (o el 97% de ellos) que hoy en día se encuentran en el panorama del mundo frontend que es lo que nos atañe a nosotros como desarrolladores. Nos permite traer el código, módulos de código abierto que han sido creados por desarrolladores del mundo entero, para que lo puedas usar en tu proyecto



## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

El ecosistema NPM está compuesto por 3 pilares; la web, la interfaz de línea de comandos o CLI y el registro.

Su uso es muy fácil: para comenzar un proyecto npm, ejecutamos 'npm init' en nuestra carpeta raíz. Una vez ejecutado nos hará unas cuantas preguntas y nos creará un archivo llamado package.json que contendrá la información más relevante de nuestro proyecto, así como los paquetes que iremos instalando con el comando 'npm install <nombre paquete>'. Junto con el comando 'npm run <orden>' serán los más usados.

Todos son ejecutados en la raíz de nuestro proyecto.

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

Para comenzar, en una terminal de consola, teclaremos ``npm init stencil``. Seleccionamos la opción 'component' y le damos un nombre a nuestro proyecto, en nuestro caso será 'stencil-course-ui'. Después de esto, sólo debemos confirmar y listo, ya tenemos lo necesario para comenzar a trabajar con Stencil. Accede a la carpeta del proyecto y con el comando ``npm install``, comenzará la instalación del core de la herramienta.

Es importante que nuestro código sea legible, bonito y limpio. Cuando escribimos código debemos hacerlo con la idea de que una persona nueva que entre en nuestro proyecto sea capaz de entenderlo rápidamente. Si trabajamos en equipo, el código debe ser homogéneo en cuanto a estilo, se podría decir que deben seguir la línea editorial impuesta. Para ello existen herramientas como Prettier y ESLint que se encargarán de ello.

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

Prettier, es una herramienta que formatea nuestro código. ¿Cómo funciona? Escribimos nuestro código y luego este es analizado y homogeneizado por la herramienta. Se descompone, se le aplica las reglas que previamente hemos configurado y se imprime de nuevo. Este proceso es opaco para ti, de esta forma tú solo debes preocuparte de tirar líneas de código que Prettier se encargará de darle consistencia .

Para integrarlo en nuestro proyecto usamos la ya mencionada línea de comandos de npm. Tecleamos ``npm install -D prettier``, luego creamos un fichero en la raíz de nuestro proyecto con el nombre ``prettierrc.json`` que contendrá las reglas que usará la herramienta englobadas entre ``{}``

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

```
{  
  "printWidth": 80,  
  "tabWidth": 2,  
  "semi": false,  
  "singleQuote": true,  
  "jsxSingleQuote": true,  
  "jsxBracketSameLine": true,  
  "arrowParens": "avoid",  
  "bracketSpacing": true  
}
```

Para finalizar modificaremos nuestro `package.json` para incluir nuestro primer guión.

```
"scripts": {  
  ...  
  "format": "prettier --write 'src/**/*.{ts,tsx,css,html,json,js,scss}'",  
}
```

Ahora el ejecutar `npm run format` sobre nuestro proyecto. El código será analizado y formateado a nuestro gusto.

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

ESlint, nos impone unas reglas de estilo. Aunque sea totalmente válido crear variables que luego, por la razón que sea, no se usan o somos desarrolladores que abusamos del método ``console.log()`` con ESLint podemos evitar que luego todo esto vaya a producción. Hemos puesto ejemplos muy simples pero esta herramienta puede conseguir que nuestro código de un salto de calidad enorme.

Instalamos con ``npm install -D eslint``. ESLint tiene distantas maneras de configurarse, en nuestro caso usaremos la guiada: ``eslint --init``. ¿Por qué? Como Stencil usa Typescript, necesitamos paquetes adicionales para que no entren en conflictos con el transpilador y la forma guiada de Eslint se encargará de ello.

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

Para ayudarnos con Stencil, instalamos una serie de paquetes adicionales que harán el trabajo 'tedioso' de configurar las herramientas y que no entren en conflicto

```
`npm install @typescript-eslint/parser @typescript-eslint/eslint-plugin eslint-plugin-react  
@stencil/eslint-plugin eslint-config-prettier -D`
```

\*si typescript no está instalado de forma global en nuestro equipo, también debe formar parte de los paquetes\*

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

Analizamos el fichero `eslinttrc.json` que se creo anteriormente y nos debería quedar algo así, si no es igual te recomiendo que lo copies como punto de partida

```
{
  "env": {
    "browser": true,
    "es2021": true
  },
  "extends": ["eslint:recommended", "plugin:@typescript-eslint/recommended", "plugin:@stencil/recommended", "prettier"],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json",
    "ecmaVersion": 12,
    "sourceType": "module"
  },
  "plugins": ["@typescript-eslint"],
  "rules": {
    "@stencil/decorators-style": ["warn", { "prop": "ignore", "method": "ignore" }],
    "@stencil/strict-mutable": "off",
    "@stencil/render-returns-host": "off"
  }
}
```

En nuestro `package.json` incluimos el guión.

```
"scripts": {
  ...
  "lint": "eslint src/**/*{.ts,.tsx}",
}
```



## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

Husky, nos provee de cierto control sobre los git hooks o ganchos git, que son scripts que se ejecutan automáticamente antes o después de que git ejecute comandos importantes como 'commit' o 'push'. Hacer esta tarea sin ayuda de Husky es algo tedioso porque nos obliga a tocar directamente en las tripas de nuestra carpeta '.git' pero como verás a continuación es fácil hacerlo con esta herramienta.

Como en ocasiones anteriores instalamos el paquete con el comando ``npm install husky -D`` y simplemente incluimos al final de nuestro ``package.json`` las siguientes líneas

```
"devDependencies": {  
  ...  
},  
"husky": {  
  "hooks": {  
    "pre-commit": "npm run format && npm run lint"  
  }  
},
```

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

Con esto estamos consiguiendo que previo a realizar el commit se revise nuestro código y cumpla los mínimos requisitos antes de ser commiteado.

Y por último, tenemos el paquete 'ling-staged'. ¿Cuál es su función? Si husky ejecutaba nuestro script contra todo el proyecto, con lint-staged ejecutaremos nuestros scripts solo contra los archivos que tengamos en nuestra staging area. ¿Para qué queremos pasar de nuevo nuestros scripts sobre ficheros que no se han tocado? ¿Porqué no usarlo sólo contra los ficheros modificados y que se encuentran preparados para ser commiteados? Esa es la función de lint-staged. Lo instala ``npm install lint-staged -D`` y al igual que con husky modificamos nuestro fichero package.json y lo dejamos tal que así

## Configuración del entorno

# npm, ESLint, Prettier, Husky & lint-staged

```
"husky": {  
  "hooks": {  
    "pre-commit": "npm run format && lint-  
staged"  
  }  
},  
"lint-staged": {  
  "src/**/*.ts,tsx,js": [  
    "npm run lint"  
  ]  
}
```

Con todo esto, ya tenemos  
preparado nuestro entorno de  
trabajo. Podemos comenzar a  
generar componentes con Stencil  
con la seguridad de que nuestro  
código será de gran calidad.

## Rafa Bernal

Programador Frontend  
{pixel developer & Stencil Advocate}  
@rbernalBer

**autentia**

