Stencil ver.3

https://stenciljs.com/

INTRODUCCIÓN A STENCIL

Introducción

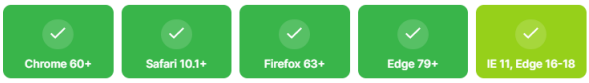
- Stencil, creado originalmente por el equipo de Ionic, es una cadena de herramientas para crear sistemas de diseño escalables y reutilizables. Genera componentes web pequeños, ultrarrápidos y 100 % basados en estándares que se ejecutan en todos los navegadores.
- Stencil no es un framework de frontend, es un compilador que genera Web Components nativos (más específicamente, Custom Elements). Stencil utiliza TypeScript, JSX y CSS para crear componentes web basados en estándares que se pueden utilizar para crear bibliotecas de componentes, sistemas de diseño y aplicaciones de alta calidad.
- Stencil combina los mejores conceptos de los framework más populares en una herramienta en tiempo de compilación simple. Las API como Virtual DOM, JSX y la representación asíncrona hacen que sea fácil crear componentes rápidos y potentes, al mismo tiempo que mantienen una compatibilidad del 100 % con los componentes web.
- Stencil genera componentes web, compatibles con los estándares, que pueden funcionar con los framework de trabajo mas populares desde el primer momento. Además, Stencil se puede usar para generar componentes nativos del framework que se pueden usar como cualquier otro componente en el framework de destino. Stencil logra esto envolviendo sus componentes web a través de la funcionalidad Stencil's Output Target.

Características

- Basado en componentes web
- Compatibilidad con TypeScript
- Definición declarativa mediante anotaciones
- Soporte JSX
- Pipeline de representación asíncrona y optimizada con VirtualDOM
- Representación previa de componentes
- Enlace de datos unidireccional
- Carga diferida de componentes simples
- Componentes libres de dependencia
- Servidor de desarrollo incorporado para recarga de módulos en caliente
- Diferencias visuales de la interfaz de usuario con capturas de pantalla
- Generación automática de la documentación de los componentes

Soporte en navegadores

- Los Custom Elements v1 se admiten de forma nativa en Chrome, Edge, Firefox y Safari (incluido iOS).
- Para navegadores sin soporte nativo, un pequeño polyfill permite usar elementos personalizados sin problemas y con poca sobrecarga de rendimiento.
- Stencil usa un cargador dinámico para cargar los polyfill solo en los navegadores que lo necesitan.



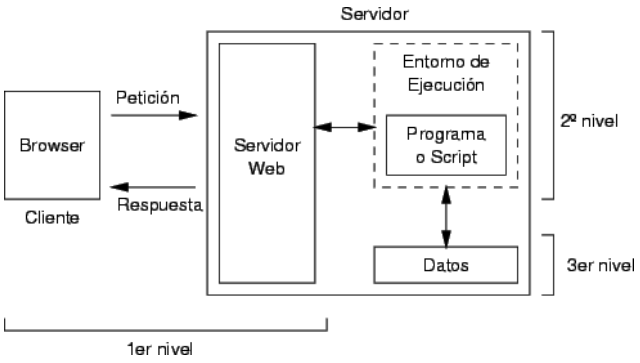
	Chrome 60+	Safari 10.1+	Firefox 63+	Edge 79+	Edge 16-18	IE 11
CSS Variables	✓	✓	✓	✓	✓	○
Custom Elements	✓	✓	✓	✓	○	○
Shadow Dom	✓	✓	✓	✓	○	○
es2017	✓	✓	✓	✓	✓	○
ES Modules	✓	✓	✓	✓	✓	○

Stencil utiliza SEMVER

- El sistema SEMVER (Semantic Versioning) es un conjunto de reglas para proporcionar un significado claro y definido a las versiones de los proyectos de software.
- El sistema SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z, donde:
  - X se denomina Major: indica cambios rupturistas
  - Y se denomina Minor: indica cambios compatibles con la versión anterior
  - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente,
  - cuando se arregla un bug se incrementa el patch,
  - cuando se introduce una mejora se incrementa el minor y
  - cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.

Arquitectura Web

Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



Aplicaciones web vs Sitios web

- Aplicaciones web
  - El objetivo principal de una aplicación es que el usuario realice una tarea. También pueden entenderse como un programa que se utiliza desde el navegador. Para crearlos, se usan los lenguajes CSS, HTML, JavaScript y se puede utilizar software gratuito de fuente abierta, como Drupal, Symfony o Meteor.
- Sitios web
  - El objetivo principal de un sitio web es entregar información. Por lo tanto, consumir contenidos es la tarea más importante hacen los usuarios en este tipo de plataformas.
  - Esta idea puede sonar confusa, ya que todos los sitios incluyen algún llamado a la acción adicional, como por ejemplo realizar un contacto o suscribirse a un newsletter. La diferencia está en que estas interacciones representan una parte pequeña y usualmente se pueden lograr solo después de guiar al usuario a través del contenido.

Enfoques

- Hay múltiples enfoques generales para crear una interfaz de usuario web moderna:
  - Sitios web:
    - Static Side Generation (SSG): Generación estática del sitio (compilados)
    - Incremental Static Regeneration (ISR): Regeneración estática incremental basada en tiempo o en caches.
  - Aplicaciones que representan la interfaz de usuario desde el servidor.
    - Server-Side Rendering (SSR)
    - Multi Page Application (MPA)
  - Aplicaciones que representan la interfaz de usuario en el cliente.
    - Client-Side Rendering (CSR)
    - Single Page Applications (SPA)
  - Aplicaciones híbridas que aprovechan los enfoques de representación de la interfaz de usuario en servidor y en cliente.

Interfaz de usuario representada en el servidor

- Una aplicación de interfaz de usuario web que se representa en el servidor genera dinámicamente el lenguaje HTML y CSS de la página en el servidor en respuesta a una solicitud del explorador. La página llega al cliente lista para mostrarse.
- Ventajas:
  - Los requisitos de cliente son mínimos porque el servidor realiza el trabajo de la lógica y la generación de páginas: Excelente para dispositivos de gama baja y conexiones de ancho de banda bajo, Permite una amplia gama de versiones de explorador en el cliente, Tiempos de carga de página iniciales rápidos, JavaScript mínimo o inexistente para ejecutar en el cliente.
  - Flexibilidad de acceso a recursos de servidor protegidos: Acceso de bases de datos, Acceso a secretos, como valores para las llamadas API a Azure Storage.
  - Ventajas del análisis de sitios estáticos, como la optimización del motor de búsqueda.
- Inconvenientes:
  - El costo del uso de proceso y memoria se centra en el servidor, en lugar de en cada cliente.
  - Lentitud, las interacciones del usuario requieren un recorrido de ida y vuelta al servidor para generar actualizaciones de la interfaz de usuario.

Interfaz de usuario representada por el cliente

- Una aplicación representada en el cliente representa dinámicamente la interfaz de usuario web en el cliente, actualizando directamente el DOM del explorador según sea necesario.
- Ventajas:
  - Permite una interactividad enriquecida que es casi instantánea, sin necesidad de un recorrido de ida y vuelta al servidor. El control de eventos de la interfaz de usuario y la lógica se ejecutan localmente en el dispositivo del usuario con una latencia mínima. Aprovecha las funcionalidades del dispositivo del usuario.
  - Admite actualizaciones incrementales, guardando formularios o documentos completados parcialmente sin que el usuario tenga que seleccionar un botón para enviar un formulario.
  - Se puede diseñar para ejecutarse en modo desconectado. Las actualizaciones del modelo del lado del cliente se sincronizan finalmente con el servidor una vez que se restablece la conexión.
  - Carga y costo del servidor reducidos, el trabajo se descarga en el cliente. Muchas aplicaciones representadas por el cliente también se pueden hospedar como sitios web estáticos.
- Inconvenientes:
  - El código de la lógica debe descargarse y ejecutarse en el cliente, lo que se agrega al tiempo de carga inicial.
  - Los requisitos de cliente pueden excluir a los usuarios que tienen dispositivos de gama baja, versiones anteriores del explorador o conexiones de ancho de banda bajo.

Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el códigos de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

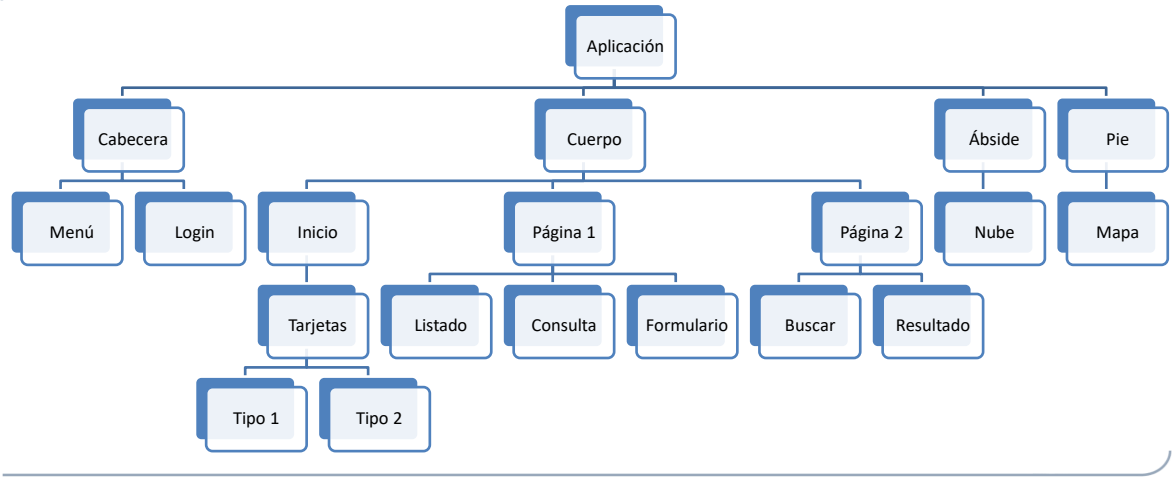
Aplicaciones web progresivas

- Las aplicaciones web progresivas (mejor conocidas como PWAs por «Progressive Web Apps») no es un nombre oficial ni formal, es solo una abreviatura utilizada inicialmente por Google para el concepto de crear una aplicación flexible y adaptable utilizando solo tecnologías web.
- Es mucho más fácil y rápido visitar un sitio web que instalar una aplicación, y se pueden compartir enviando un enlace. Por otro lado, las aplicaciones nativas están mejor integradas con el sistema operativo y, por lo tanto, ofrecen una experiencia más fluida para los usuarios, se puede instalar una aplicación nativa para que funcione sin conexión, y los usuarios pueden tocar sus íconos para acceder fácilmente a sus aplicaciones favoritas sin tener que navegar. Las PWA nos brindan la capacidad de crear aplicaciones web que se comportan como nativas.
- Son aplicaciones web que utilizan APIs y funciones emergentes del navegador web junto a una estrategia tradicional de mejora progresiva para ofrecer una aplicación nativa, con la experiencia de usuario de aplicaciones web multiplataforma. Se puede pensar que PWA es similar a AJAX u otros patrones similares, son aplicaciones web desarrolladas con una serie de tecnologías específicas y patrones estándar que les permiten aprovechar las funciones de las aplicaciones nativas y web, aunque no son un estándar formalizado.
- Las PWA deben ser detectables, instalables, enlazables, independientes de la red, progresivas, reconectables, responsivas y seguras.

Componentes

- Los sistemas de hoy en día son cada vez más complejos, deben ser construidos en tiempo récord y deben cumplir con los estándares más altos de calidad.
- El paradigma de ensamblar componentes y escribir código para hacer que estos componentes funcionen entre si se conoce como Desarrollo de Software Basado en Componentes.
- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente. Sigue un modelo de composición jerárquico con forma de árbol de componentes. La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Los componentes establecen un cauce bien definido de entrada/salida para su comunicación con otros componentes.

Árbol de componentes



Web Components

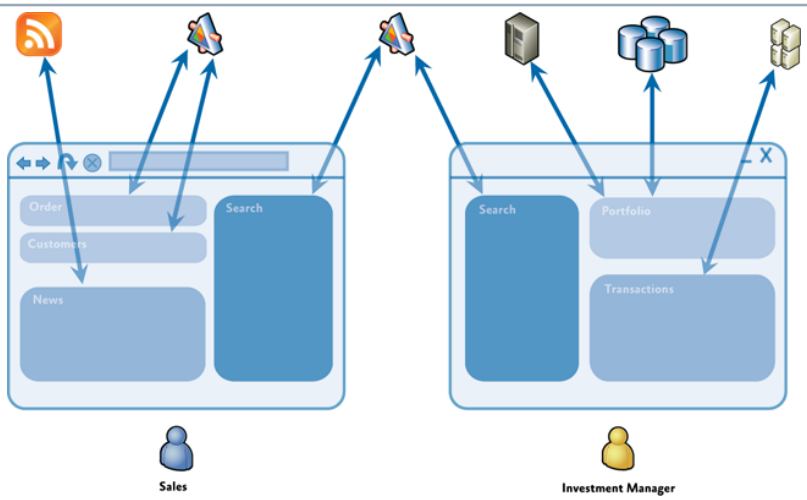
- Los Web Components nos ofrecen un estándar que va enfocado a la creación de todo tipo de componentes utilizables en una página web, para realizar interfaces de usuario y elementos que nos permitan presentar información (o sea, son tecnologías que se desarrollan en el lado del cliente). Los propios desarrolladores serán los que puedan, en base a las herramientas que incluye Web Components crear esos nuevos elementos y publicarlos para que otras personas también los puedan usar.
- Los Web Components son una reciente incorporación al HTML5 que, si siguen evolucionando al ritmo al que lo están haciendo, pueden suponer el mayor cambio en el mundo web en años y solucionar de golpe varios problemas históricos de HTML.
- El estándar, en proceso de definición, en realidad, se compone de 4 subelementos complementarios, pero independientes entre si:
  - Custom Elements http://w3c.github.io/webcomponents/spec/custom/
  - Shadow DOM https://www.w3.org/TR/shadow-dom/
  - Templates Elements https://www.w3.org/TR/html5/scripting-1.html#the-template-element
  - ~~HTML Imports http://w3c.github.io/webcomponents/spec/imports/~~
- Cualquiera de ellos puede ser usado por separado, lo que hace que la tecnología de los Web Elements sea, además de muy útil, muy flexible.

Web Components

- **Custom elements** (elementos personalizados): Un conjunto de APIs de JavaScript que permiten definir elementos personalizados y su comportamiento, que despues pueden ser utilizado como se deseé en la interfaz del usuario.
- **Shadow DOM**: Un conjunto de APIs de JavaScript para fijar un árbol DOM "sombra" encapsulado a un elemento, que es renderizado por separado del documento DOM principal, y controlando funcionalidad asociada. De esta forma, se pueden mantener características de un elemento en privado, así puede tener el estilo y los scripts sin miedo de colisiones con otras partes del documento.
- **HTML Templates** (plantillas HTML): Los elementos <template> y <slot> permiten escribir plantillas de marcado que no son desplegadas en la página renderizada. Éstas pueden ser reutilizadas en múltiples ocasiones como base de la estructura de un elemento personalizado.
- **HTML Import** (descartada): ~~es la posibilidad de incluir un HTML dentro de otro mediante un tag <link rel="import" href="include.html">.~~

https://developer.mozilla.org/en-US/docs/Web/Web_Components

Patrón Composite View



Micro Frontends

- El término Micro Frontends apareció por primera vez en ThoughtWorks Technology Radar a finales de 2016. Extiende los conceptos de los micro servicios al mundo del frontend. La tendencia actual es crear una aplicación de navegador potente y rica en características, también conocida como “single page app”, que se asiente sobre una arquitectura de microservicio. Con el tiempo, la capa de frontend, a menudo desarrollada por un equipo independiente, crece y se vuelve más difícil de mantener. Eso es lo que llamamos una Interfaz Monolítica.
- La idea detrás de Micro Frontends es pensar en un sitio web o aplicación web como una composición de características que son propiedad de equipos independientes. Cada equipo tiene un área de negocio definida o misión de la que se preocupa y se especializa. Un equipo es cross functional y desarrolla sus características end-to-end, desde la base de datos hasta la interfaz de usuario.
- Los Custom Elements, el aspecto de interoperabilidad de las especificaciones de Web Components, son una buena primitiva para la integración en el navegador. Cada equipo construye sus componentes usando la tecnología web de su elección y lo envuelve dentro de un Custom Element. La especificación DOM de dichos componentes (nombre de etiqueta, atributos y eventos) actúa como el contrato o API pública para otros equipos. La ventaja es que pueden usar el componente y su funcionalidad sin tener que conocer la implementación. Solo tienen que ser capaces de interactuar con el DOM. El equipo de producto decide la composición de la aplicación, cómo ensamblar una página con componentes que pertenecen a diferentes equipos.

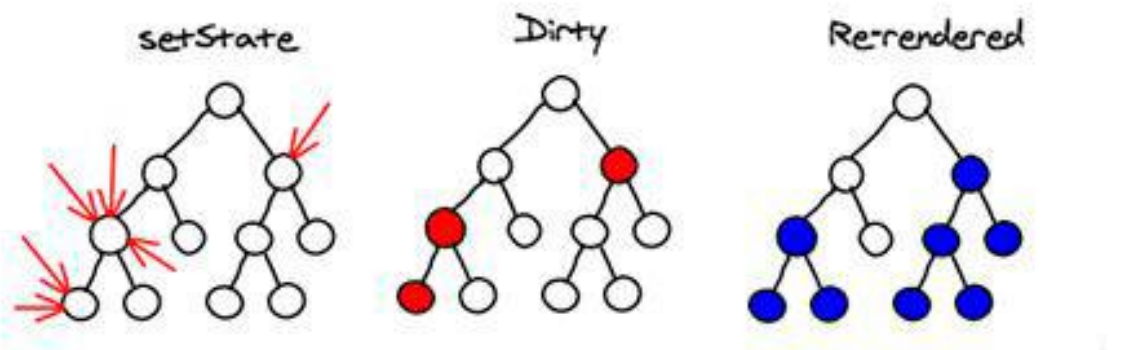
WebAssembly (wasm)

- WebAssembly es un nuevo tipo de código que puede ser ejecutado en navegadores modernos. Es un lenguaje de bajo nivel, similar al lenguaje ensamblador, con un formato binario compacto que se ejecuta con rendimiento casi nativo y provee un objetivo de compilación para lenguajes como C/C++ y Rust que les permite correr en la web. También está diseñado para correr a la par de JavaScript, permitiendo que ambos trabajen juntos.
- WebAssembly tiene grandes implicaciones para la plataforma web, provee una forma de ejecutar código escrito en múltiples lenguajes en la web a una velocidad casi nativa, con aplicaciones cliente corriendo en la web que anteriormente no podrían haberlo hecho.
- WebAssembly esta diseñado para complementar y ejecutarse a la par de JavaScript, usando las APIs WebAssembly de JavaScript, se pueden cargar módulos de WebAssembly en una aplicación JavaScript y compartir funcionalidad entre ambos. Esto permite aprovechar el rendimiento y poder de WebAssembly con la expresividad y flexibilidad de JavaScript en las mismas aplicaciones.
- En diciembre de 2019, se publicaron la especificaciones WebAssembly Core Specification, WebAssembly Web API y WebAssembly JavaScript Interface como recomendaciones del W3C.

Virtual DOM

- El secreto de ReactJS para tener un rendimiento muy alto, era que implementa algo denominado Virtual DOM: en vez de renderizar (repintar) todo el DOM en cada cambio, que es lo que normalmente se hace, ReactJS hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las propiedades de la versión actual del DOM y así aplicar cambios exclusivamente en las partes que varían. El algoritmo diferencial se encarga de actualizar el nuevo DOM comparándolo con el previo.
- Stencil a tomado la idea del Virtual DOM de React.
- En lugar de generar directamente elementos HTML y modificar el DOM, que es una operación muy lenta, los componentes de Stencil generan (renderizan) un modelo del DOM en memoria. Una vez generado el Virtual DOM completo, Stencil se encarga de buscar las diferencias entre el DOM real frente al virtual y realizar únicamente las modificaciones necesarias sobre el DOM real.
- Esto permite un estilo de programación mas similar al SSR que al del CSR.

Virtual DOM



HERRAMIENTAS DE DESARROLLO

IDEs

- Visual Studio Code - http://code.visualstudio.com/
  - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
  - Visual Studio Code for the Web provides a free, zero-install Microsoft Visual Studio Code experience running entirely in your browser: https://vscode.dev
- StackBlitz - https://stackblitz.com
  - The online IDE for web applications. Powered by VS Code and GitHub.
- CodeSandbox - https://codesandbox.io/
  - Create, share, and get feedback with collaborative sandboxes.
- IntelliJ IDEA - https://www.jetbrains.com/idea/
  - Capable and Ergonomic Java * IDE
- Webstorm - https://www.jetbrains.com/webstorm/
  - Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: https://git-scm.com/
- Verificar desde consola de comandos:
  - git

Node.js: Entorno en tiempo de ejecución

- Descargar e instalar: https://nodejs.org
- Verificar desde consola de comandos:
  - node --version

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
  - npm update -g npm
- Verificar desde consola de comandos:
  - npm --version
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

Generación del esqueleto de aplicación

- Configurar un nuevo proyecto es un proceso complicado y tedioso, con tareas como:
  - Crear la estructura básica de archivos y bootstrap
  - Configurar SystemJS o WebPack para transpilar el código
  - Crear scripts para ejecutar el servidor de desarrollo, tester, publicación, …
- Disponemos de diferentes opciones de asistencia:
  - Proyectos semilla (seed) disponibles en github
  - Herramienta oficial de gestión de proyectos, Stencil CLI: Creada por el equipo de Stencil, es una Command Line Interface que permite generar proyectos y componentes desde consola, así como ejecutar un servidor de desarrollo o lanzar los tests de la aplicación (la instalación es opcional y no se recomienda).

Stencil CLI

- Creada por el equipo de Stencil, es una Command Line Interface que permite generar proyectos y componentes desde consola, así como ejecutar un servidor de desarrollo o lanzar los tests de la aplicación (la instalación es opcional y no se recomienda).
- Dependencia única: solo hay una dependencia de compilación. Utiliza Rollup, TypeScript, Jest, Puppeteer y otros proyectos increíbles, pero ofrece una experiencia coherente combinando todos ellos.
- No requiere configuración: no se necesita configurar nada. Se maneja una configuración razonablemente buena de compilaciones tanto de desarrollo como de producción para poderse concentrar en la escritura de código.
- Stencil requiere una versión LTS reciente de NodeJS y npm.

Creación y puesta en marcha

- Crear un nuevo proyecto

  $ npm init stencil

  – Esto creará la carpeta del proyecto con un esqueleto de proyecto ya montado según la última versión y todo el trabajo sucio de configuración de tests, lint y typescript, etc.
    - component: Colección de componentes web que se pueden usar en cualquier lugar
      – https://github.com/ionic-team/stencil-component-starter
    - app [community]: Crea una aplicación o un sitio web con Stencil
      – https://github.com/stencil-community/stencil-app-starter
    - ionic-pwa [community]: Crea una Ionic PWA con diseño de pestañas y rutas

  $ cd proyecto && npm install

  – Cambiar al directorio creado e instalar las dependencias
- Servidor de desarrollo

  $ npm start (npx stencil build --dev --watch --serve)

  – Esto lanza tu app en la URL http://localhost:3333 y actualiza el contenido cada vez que guardas algún cambio.

Otros comandos

$ npm run generate <sub-folder>

– Inicia el generador de componentes interactivos. Se puede especificar una o más subcarpetas para generar el componente. Se puede indicar que ficheros se deben generar:
  - (*) Stylesheet (.css)
  - (*) Spec Test (.spec.tsx)
  - (*) E2E Test (.e2e.ts)

$ npm run build

– Crea una versión lista para producción de los componentes. Los componentes generados en este paso no están destinados a usarse en el servidor de desarrollo local, sino dentro de un proyecto que consume los componentes.

$ npm test

– Ejecuta las pruebas del proyecto. Stencil CLI ha creado pruebas unitarias y de extremo a extremo al crear la estructura del proyecto y al generar los componentes.

GIT

- Preséntate a Git
  - git config --*global* user.name "Your Name Here“
  - git config --*global* user.email your_email@youremail.com
- Crea un repositorio central
  - https://github.com/
- Conecta con el repositorio remoto
  - git remote add origin https://github.com/username/myproject.git
  - git push -u origin master
- Actualiza el repositorio con los cambios:
  - git commit -m "first commit"
  - git push
- Para clonar el repositorio:
  - git clone https://github.com/username/myproject.git *local-dir*
- Para obtener las últimas modificaciones:
  - git pull

TypeScript

TypeScript

- El lenguaje TypeScript fue ideado por Anders Hejlsberg, autor de Turbo Pascal, Delphi, C# y arquitecto principal de .NET, como un supraconjunto de JavaScript que permitiese utilizar tipos estáticos y otras características de los lenguajes avanzados como la Programación Orientada a Objetos.
- TypeScript es un lenguaje Open Source basado en JavaScript y que se integra perfectamente con otro código JavaScript, solucionando algunos de los principales problemas que tiene JavaScript:
  - Falta de tipado fuerte y estático.
  - Falta de "Syntactic Sugar" para la creación de clases
  - Falta de interfaces (aumenta el acoplamiento ya que obliga a programar hacia la implementación)
  - Módulos (parcialmente resuelto con require.js, aunque está lejos de ser perfecto)
- TypeScript es un lenguaje con una sintaxis bastante parecida a la de C# y Java, por lo que es fácil aprender TypeScript para los desarrolladores con experiencia en estos lenguajes y en JavaScript.

TypeScript

- Lo que uno programa con TypeScript, tras grabar los cambios, se convierte en JavaScript perfectamente ejecutable en todos los navegadores actuales (y antiguos), pero habremos podido:
  - abordar desarrollos de gran complejidad,
  - organizando el código fuente en módulos,
  - utilizando características de auténtica orientación a objetos, y
  - disponer de recursos de edición avanzada del código fuente (Intellisense, compleción de código, "snippets", etc.).
- TypeScript amplia la sintaxis del JavaScript con la definición y comprobación de:
  - Tipos básicos: booleans, number, string, Any y Void.
  - Clases e interfaces
  - Herencia
  - Genéricos
  - Módulos
  - Anotaciones

TypeScript

- El TypeScript se traduce (“transpila”, translate+compiler, compilar un lenguaje de alto nivel a otro de alto nivel o de niveles similares) a JavaScript cumpliendo el estándar ECMAScript totalmente compatible con las versiones existentes.
- De hecho, el “transpilador” deja elegir la versión ECMAScript del resultado, permitiendo adoptar la novedades mucho antes de que los navegadores las soporten: para cambiar de versión basta con configurar y volver a transpilar, dejando el resultado en la versión mas ampliamente difundida.
- El equipo de TypeScript provee de una herramienta de línea de comandos para hacer esta compilación, se puede instalar con el siguiente comando:
  - npm install -g typescript
- Podemos transpilar manualmente escribiendo
  - tsc helloworld.ts
- Los diversos “plug-in” se pueden descargar en: https://www.typescriptlang.org

Comandos tsc

- Compilación continua:
  - tsc –w *.ts
  - tsc --watch *.ts
- Compilación de múltiples ficheros de entrada en un fichero único de salida
  - tsc --outFile file.js *.ts
- Compilación del proyecto:
  - -p DIRECTORY, --project DIRECTORY
- Control de directorios
  - --rootDir LOCATION ← Ficheros de entrada
  - --outDir LOCATION ← Ficheros de salida
- Directorio base de las rutas de los módulos:
  - --baseUrl
- Creación del fichero de configuración tsconfig.json:
  - --init

tsconfig.json

- El archivo tsconfig.json especifica los archivos raíz y las opciones del compilador necesarias para compilar el proyecto. La presencia de un archivo tsconfig.json en un directorio indica que el directorio es la raíz de un proyecto de TypeScript.

```
{
  "compilerOptions": {
    "target": "ES2015",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "paths": {
      "@my/module": ["src/my/module/index.ts"]
    }
  },
  "$schema": "https://json.schemastore.org/tsconfig",
  "display": "Recommended"
}
```

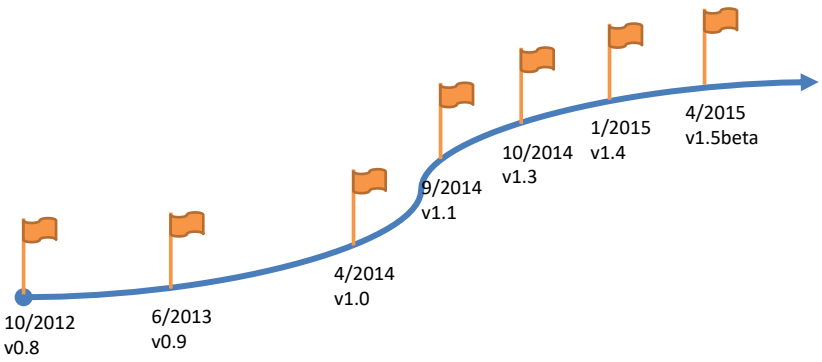

Soportado por IDE's/Utilidades

- Visual Studio 2012 … – native (+msbuild)
- Visual Studio Code
- ReSharper – included
- Sublime Text 2/3 – official plugin as a part of 1.5 release
- Online Cloud9 IDE
- Eclipse IDE
- IntelliJ IDEA
- Grunt, Maven, Gradle plugins

Eclipse

- Requiere tener instalado Node.js y TypeScript
- Abrir Eclipse
- Ir a Help → Install New Software
  - Add the update site: http://eclipse-update.palantir.com/eclipse-typescript/
  - Seleccionar e instalar TypeScript
  - Re arrancar Eclipse
- Opcionalmente, con el botón derecho en el proyecto seleccionar Configure → Enable TypeScript Builder
- En Project → Properties → Builders
  - Add new "Program" builder
  - Location: ...\tsc
  - Working Directory: ${project_loc}
  - Arguments: --source-map --outFile build.js **/*.ts
- En Project → Properties → Build Automatically

Historia



Compatibilidad con ES6

Feb 2015

				9%	
Compilers/polyfills					
63%	75%	31%	16%	9%	21%
Traceur	Babel + core-js[1]	Closure	JSX[2]	Type-Script	es6-shim
0/2	1/2	0/2	0/2	0/2	0/2
3/6	5/6	4/6	0/6	3/6	0/6
4/5	3/5	2/5	3/5	3/5	0/5
10/10	10/10	2/10	0/10	0/10	0/10
6/6	6/6	5/6	3/6	3/6	0/6
7/7	7/7	4/7	0/7	0/7	0/7
2/4	4/4	2/4	0/4	2/4	0/4
2/2	2/2	2/2	2/2	1/2	0/2

Today (TS 1.5)

				52%	
Compilers/polyfills					
58%	71%	31%	18%	52%	17%
Traceur	Babel + core-js[1]	Closure	JSX[2]	Type-Script + core-js	es6-shim
0/2	1/2	0/2	0/2	0/2	0/2
3/6	5/6	4/6	0/6	4/6	0/6
4/5	4/5	2/5	3/5	3/5	0/5
12/12	12/12	2/12	1/12	2/12	0/12
6/6	6/6	4/6	5/6	6/6	0/6
8/8	8/8	5/8	1/8	2/8	0/8
2/4	4/4	2/4	0/4	4/4	0/4

https://kangax.github.io/compat-table/es6/#typescript

Tipos de datos

- Boolean: var isDone: boolean = false;
- Number: var height: number = 6;
- String: var name: string = 'bob';
- Enum: enum Color {Red, Green, Blue};
  var c: Color = Color.Green;
- Array: var list:number[] = [1, 2, 3];
  var list:Array<number> = [1, 2, 3];
- Object: no es number, string, boolean, symbol, null, o undefined.
- Any, Unknown: var notSure: any = 4;
  notSure = 'maybe a string instead';
  notSure = false; // okay, definitely a boolean
  var list:any[] = [1, true, 'free'];
- Void, Never: function warnUser(msg: string): void { alert(msg); }
- Null, Undefined
- Funciones, Clases e Interfaces

Tipos especiales

- El tipo **any** representa cualquier valor, desactiva la verificación de tipos en aquellos casos una dependencia externa impide establecer tipos concretos.
- El tipo **unknown** representa cualquier valor. Es similar al tipo any, pero es más seguro porque no es legal hacer nada con un valor unknown.
- El tipo **void** es un pseudo tipo que representa la no devolución de valor por parte de una función o método. Solo se puede utilizar como tipo de retorno.
- El tipo **never** representa valores que nunca se observan. En un tipo de retorno de funciones, esto significa que la función lanza una excepción o termina la ejecución del programa.

Definiciones

- Variables globales (no deberían usarse para evitar el acoplamiento):
  ```
  var str: string = 'Inicio';
  ```
- Variables locales:
  ```
  let x, y, z: number; //  undefined
  ```
- Variables constantes:
  ```
  const LEVEL: number = 4;
  ```
- Inferencia de tipo:
  ```
  let str = 'Inicio'; // string
  let x = 1, y = 1; // number
  ```
- Tuplas:
  ```
  let tupla: [number, string] = [1, 'cadena'];
  let x : number = tupla[0];
  ```
- Alias a los tipos:
  ```
  type Tupla = [number, string];
  let tupla: Tupla = [1, 'cadena'];
  ```

Control de tipos

- Las anotaciones de tipo en TypeScript son formas ligeras de registrar el contrato previsto de la variable o función, permitiendo en la compilación la validación de su correcto uso.

```
1  var myNumber: number = 5;
2
3  var myString = "5";
4
5  var strResult = myNumber + myString;
6
7  var object = { field: 1, field2: myString }
8
9  var err1 = object.field3;
10
11 var err2 = myNumber + object;
```


Sintaxis ampliada (ES6)

- Cadenas (Interpolación y múltiples líneas)

```
let msg = `El precio de ${producto} es
  de ${unidades * precio} euros`;
```

- Bucle foreach:

```
for (let i of [10, 20, 30]) { window.alert(i); }
```

- Destructing

```
let x = 1;
let y = 2;
[x, y] = [y, x];

let point = {x: 1, y: 2};
{x, y} = point;
```

- Auto nombrado:

```
let x = 1;
let y = 2;
let point = {x, y}; // {x: x, y: y}
```

Funciones con tipos

- Tipos para los parámetros y el valor de retorno:
```
function add(x: number, y: number): number { return x+y; }
```
- Opcionales y Valores por defecto: Se pueden definir valores por defecto a los parámetros en las funciones o marcarlos opcionales (undefined).
```
function f(opcional?: any, valor = 'foo') {
   if(opcional) {
      :
   }
   :
};
```
- Resto de los parámetros: Convierte una lista de parámetros en un array (Any por defecto).
```
function f (x: number, y: number, ...a: any[]): number {
   return (x + y) * a.length;
}
f(1, 2, 'hello', true, 7) === 9
```


Funciones con tipos

- Se puede utilizar la desestructuración de parámetros para descomprimir convenientemente los objetos proporcionados como argumento en una o más variables locales en el cuerpo de la función.
```
function sum({ a, b, c }) {
  console.log(a + b + c);
}
let obj = { a: 10, b: 3, c: 9 }
sum(obj);
```
- Operador de propagación: Convierte un array, objeto o cadena en una lista de parámetros.
```
let str = 'foo';
let chars = [ ...str ]; // [ 'f', 'o', 'o' ]
let punto = { x: 10, y: 5 } // fn(...punto) equivale a fn(punto.x, punto.y)
```

Sobrecargas de funciones

- Algunas funciones de JavaScript se pueden llamar con una variedad en número y tipos de argumentos. En TypeScript, podemos especificar que una función se puede llamar de diferentes formas escribiendo firmas de sobrecarga. Para hacer esto, se escribe las firmas de la función, seguidas del cuerpo de la función, que debe contemplar la variabilidad de los parámetros:

```
function makeDate(timestamp: number): Date;
function makeDate(m: number, d: number, y: number): Date;
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
  if (d !== undefined && y !== undefined) {
    return new Date(y, mOrTimestamp, d);
  } else {
    return new Date(mOrTimestamp);
  }
}
const d1 = makeDate(12345678);
const d2 = makeDate(5, 5, 5);
const d3 = makeDate(1, 3);
```


Expresiones Lambda (Arrow functions)

- Las expresiones Lambda o arrow functions (en algunos lenguajes) son funciones anónimas.
- El operador => dispone de diferente firmas para la definición abreviada de las funciones anónimas:
  - Parámetro único: item => …
  - Sin parámetros: () => …
  - Varios parámetro: (a, b) => …
  - Cuerpo como expresión (return implicito): (a, b) => a + b
  - Cuerpo con bloque de instrucciones (return explicito): () => { …; return rslt; }

```
let rslt = data.filter(item => item.value > 0);
// equivale a: let rslt = data.filter(function (item) { return item.value > 0; });
data.forEach(elem => {
   console.log(elem);
   // …
});
let fn = (num1, num2) => num1 + num2;
pairs = evens.map(v => ({ even: v, odd: v + 1 }));
```

Enumeraciones

- Las enumeraciones permiten a un desarrollador definir un conjunto de constantes con nombre. El uso de enumeraciones puede facilitar la documentación de la intención o restringir el conjunto de casos distintos.
- Las enumeraciones son una de las pocas características que tiene TypeScript que no es una extensión a nivel de tipo de JavaScript, desaparecen como tales en tiempo ejecución.
- TypeScript proporciona enumeraciones tanto numéricas (por defecto, base 0 con autoincremento para los miembros sin valor explicito) como basadas en cadenas.

```
enum Direction { Up = 1, Down, Left, Right, }

let direction: Direction = Direction.Left;
if(direction !== Direction.Up) {
   console.log(direction)
}

enum LogLevel {
  ERROR = 'error',
  WARN = 'warn',
  INFO = 'info',
  DEBUG = 'debug',
}
```


Tipos Unión

- El sistema de tipos de TypeScript permite crear nuevos tipos combinando los existentes.
- Un tipo de unión es un tipo formado por dos o más tipos concatenados con | y que representan valores que pueden ser cualquiera de esos tipos.
- Nos referimos a cada uno de estos tipos como miembros de la unión.
- Los miembros de la unión pueden ser valores concretos como cadenas, objetos, …
- TypeScript solo permitirá una operación si es válida para todos los miembros de la unión.
- La solución es estrechar la unión, que ocurre cuando TypeScript puede deducir un tipo más específico para un valor basado en la comprobación de tipos: typeof, Array.isArray, …

```
type ModoCRUD = 'list' | 'add' | 'edit' | 'view' | 'delete';
let modo: ModoCRUD = 'list';

function f(x: number | number[], factor: number = 1) {
  if(typeof x === "number") {
    return x * factor;
  } else {
    let sum = 0;
    x.forEach(item => {
      sum += item;
    })
    return sum * factor;
  }
}

console.log(f(10, 2))
console.log(f([10, 20, 30]))
```

Null y Undefined (v2.1)

- En TypeScript, tanto el nulo como el sin definir tienen sus propios tipos denominados undefined y null respectivamente.
- Al igual que el vacío, no son muy útiles por su cuenta.

```
let u: undefined = undefined;
let N: null = null;
```

- Por defecto, null y undefined son subtipos de todos los demás tipos, por lo que se puede asignar null o undefined a cualquier otro tipo.
- Cuando se utiliza el flag --strictNullChecks, null y undefined sólo se pueden asignar a void y sus respectivos tipos.
  - Esto ayuda a evitar muchos errores comunes (error del billón de dólares: null pointer exception).
  - Si se desea utilizar en un tipo el valor nulo o indefinido, se puede definir un tipo unión: tipo | null | undefined.

Prevención de errores

- Operador para accesos de propiedad opcionales (resolución temprana de nulos)

```
let x = foo?.bar.baz();
```

- Operador de aserción no nulo (omite la comprobación de posible nulo)

```
let x = cad!.toUpperCase();
```

- Operador de fusión nula (valor por defecto para los nulos)

```
let x = foo ?? bar();
```

- En las veces que hay un valore sobre el que TypeScript no puede conocer su tipo, el operador as puede resolverlo.

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

- Funciones de aserción

```
function assert(condition: any, msg?: string): asserts condition {
  if (!condition) { throw new AssertionError(msg) }
}
assert(typeof x === "number");
```

Objetos JavaScript

```
1  function Person(age, name, surname) {
2      this.age = age;
3      this.name = name;
4      this.surname = surname;
5  }
6
7  Person.prototype.getFullName = function () {
8      return this.name + " " + this.surname;
9  }
10
11 var cadaver = new Person(60, "John", "Doe");
12
13 var boy = Person(10, "Vasya", "Utkin"); //bad
```


Clases

- TypeScript ofrece soporte casi completo para las clases de orientación a objetos (como la sobrecarga de métodos y los aspectos en tiempo de ejecución).
- Permite implementar:
  - Atributos, métodos y propiedades
  - Encapsulación con los modificadores de acceso:
    - **public** ← ***por defecto***
    - private
    - protected *(a partir de 1.3)*
  - Atributos de solo lectura con el modificador readonly (deben inicializarse en su declaración o en el constructor)
  - Sobre escritura de métodos con el modificador Override
  - Miembros de clase, con el modificador static (name, length y call no son validos)
  - Herencia, implementación de interfaces, genéricos
  - Clases y miembros abstractos con el modificador abstract

Clases

```
class Persona {
  readonly MAYORIA_EDAD = 18;
  private id: number;
  private nombre: string;
  private apellidos: string;
  protected edad: number;
  constructor(id: number, nombre: string, apellidos: string, edad: number) {
    this.id = id;
    this.nombre = nombre;
    this.apellidos = apellidos;
    this.edad = edad;
  }
  public get Id(): number { return this.Id }
  public getNombreCompleto(): string { return `${this.nombre} ${this.apellidos}`}
  public esMayorDeEdad(): boolean { return this.edad >= this.MAYORIA_EDAD }
  public cumpleAños(): void { if(!this.esMayorDeEdad()) this.edad++ }
}

let boy = new Persona(1, 'Peter', 'Pan', 10)
let cadaver = Persona(2, 'Cruela', 'de Evil', 101) // Error de compilación
```


Clases

- El acceso a los miembros dentro de la clase debe realizarse siempre a través de this.
- TypeScript ofrece una sintaxis especial para convertir un parámetro del constructor en un atributo de la clase con el mismo nombre y valor. Se crean anteponiendo al parámetro uno de los modificadores de visibilidad: public, private, protected o readonly. El atributo resultante se define con dicho modificador:

```
constructor(private id: number, private nombre: string, private apellidos: string, protected edad:
      number) { }
```

- Un inicializador puede dirigir la creación de las clase e inicializar los atributos de clase, tal y como pasa con los métodos de clase (static) no puede acceder a los miembros de instancia:

```
private static list: Array<Persona>;
static {
  this.list = []
  this.list.push(new Persona(1, 'Peter', 'Pan', 10))
  this.list.push(new Persona(2, 'Cruela', 'de Evil', 101))
}
```

Propiedades

- Las propiedades aportan toda una serie de ventajas:
  - Externamente se comportan como un atributo público.
  - Internamente están compuestas por dos métodos, con los descriptores de acceso get (función) y set (procedimiento), que controlan como entra y sale la información.
- TypeScript tiene algunas reglas especiales para los accesores:
  - Getters y setters deben tener el mismo tipo y la misma visibilidad de miembro.
  - Si no se especifica el tipo del parámetro setter, se deduce del tipo de retorno del getter.
  - Las propiedades de solo lectura solo disponen del método get.
  - No tienen que tener correspondencia directa con los atributos.

```
private nombre: string;
public get Nombre(): string {
  return this.nombre.toUpperCase();
}
public set Nombre(valor: string) {
  if (this.nombre === valor) return;
  if (!valor)
    throw new RangeError('null value');
  this.nombre = valor;
  this.onPropertyChange('Nombre');
}
if(boy.Nombre === '') {
    boy.Nombre = 'Peter'
 }
```


Compila a ES5

```
var Persona = /** @class */ (function () {
    function Persona(id, nombre, apellidos, edad) {
        this.MAYORIA_EDAD = 18;
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.edad = edad;
    }
    Object.defineProperty(Persona.prototype, "Id", {
        get: function () { return this.Id; },
        enumerable: false,
        configurable: true
    });
    Persona.prototype.getNombreCompleto = function () { return "".concat(this.nombre, " ").concat(this.apellidos); };
    Persona.prototype.esMayorDeEdad = function () { return this.edad >= this.MAYORIA_EDAD; };
    Persona.prototype.cumpleAños = function () { if (!this.esMayorDeEdad()) this.edad++; };
    return Persona;
}());
```

También a ES6 o superior

```
class Persona {
   constructor(id, nombre, apellidos, edad) {
      this.MAYORIA_EDAD = 18;
      this.id = id;
      this.nombre = nombre;
      this.apellidos = apellidos;
      this.edad = edad;
   }
   get Id() { return this.Id; }
   getNombreCompleto() { return `${this.nombre} ${this.apellidos}`; }
   esMayorDeEdad() { return this.edad >= this.MAYORIA_EDAD; }
   cumpleAños() { if (!this.esMayorDeEdad()) this.edad++; }
}
```


Herencia

- Las clases pueden extender una clase base (solo una). Una clase derivada tiene todos los miembros de su clase base y puede definir miembros adicionales o sobre escribir métodos heredados.

```
class Adulto extends Persona {
  readonly EDAD_JUBILACION;
  constructor(id: number, nombre: string, apellidos: string, edad: number) {
    super(id, nombre, apellidos, edad)
    this.EDAD_JUBILACION = 67;
  }
  public sePuedeJubilar(): boolean { return this.edad >= this.EDAD_JUBILACION }
  public override cumpleAños(): void { if (this.esMayorDeEdad()) this.edad++; else super.cumpleAños(); }
}
```

- Si la clase base tiene un costructor, el heredero deberá llamar super(); en el cuerpo de su constructor antes de usar cualquier miembro this. Acepta la sobre escritura de métodos pero deben ir marcados con Override si está activada directiva -noImplicitOverride. La clase base puede ser abstracta (no instanciable), en cuyo caso puede tener métodos abstractos:

```
abstract class Persona {
  abstract getTitulo(): string;
```

Genéricos

- Las clases e interfaces genéricas tienen una lista de parámetros de tipo genérico entre corchetes angulares (<>) después del nombre.

```
class Elemento<T> {
  constructor(private key: T, private value: string) {}
  public get Key() { return this.key; }
  public get Value() { return this.value; }
}
let genero = new Elemento<string>('M', 'Masculino')
let provincia = new Elemento(28, 'Madrid')
let tipo: string = genero.Key
let id: number = provincia.Key;
```

- Se puede inferir los tipos de los parámetros de tipo si aparecen en el constructor.
- Se puede usar una restricción para limitar los tipos que puede aceptar un parámetro de tipo.

```
class Elemento<T extends number | string> {
```


Funciones genéricas

- Las funciones genéricas se implementan declarando parámetros de tipo en la firma de la función:

```
function firstElement<Type>(arr: Type[]): Type | undefined { return arr[0]; }
const s = firstElement(["a", "b", "c"]); // s is of type 'string'
const n = firstElement([1, 2, 3]); // n is of type 'number'
const u = firstElement([]); // u is of type undefined
```

- El tipo es inferido, elegido automáticamente, por TypeScript. Se puede inferir de los parámetros de entrada y de tipo de retorno.

```
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {
    return arr.map(func);
}
const parsed = map(["1", "2", "3"], (n) => parseInt(n)); // Input is 'string' and Output is 'number[]'
```

- Se puede usar una restricción para limitar los tipos que puede aceptar un parámetro de tipo.

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
```

Interfaces

- Al igual que otros lenguajes orientados a objetos, TypeScript permite la definición e implementación de interfaces pero con matices:

```
interface Pingable {
  ping(): void;
}
class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}
```

- Los tipos interface desaparecen en tiempo de ejecución (no se pueden comprobar con instanceof o similares), son restricciones para la compilación. Las clases no están obligadas a implementarlos pero si a cumplirlos, es solo una verificación de que la clase se puede tratar como el tipo de interfaz.

Interfaces: Propiedades solo lectura y opcionales

```
interface SquareConfig {
    readonly id: number;
    color?: string;
    width?: number;
}
function createSquare(config: SquareConfig): { color: string; area: number } {
    let newSquare = { color: 'white', area: 100 };
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}
let mySquare = createSquare({ id: 1, color: 'black' });
```

Interfaces: Como prototipos

- Usando interfaces:

```
interface LabelledValue {
  label: string;
}
function printLabel(labelledObj: LabelledValue) { console.log(labelledObj.label); }
let myObj = {size: 10, label: 'Size 10 Object'};
printLabel(myObj);
```

- Definición en línea, sin interfaces:

```
function printLabel(labelledObj: {label: string}) {  console.log(labelledObj.label); }
let myObj = {size: 10, label: 'Size 10 Object'};
printLabel(myObj);
```


Interfaces: Funciones y Arrays

- Prototipos o firmas de Funciones

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
let mySearch: SearchFunc = function (source: string, subStr: string) {
  return source.search(subStr) != -1;
}
```

- Prototipos de Arrays o colecciones

```
interface StringArray {
  [index: number]: string;
}
interface StringDictionary {
  [index: string]: string; // Para poder usar la sintaxis "indexado" (obj["key"]) además de la de "punto" (obj.key)
}
let myArray: StringArray = ['Bob', 'Fred'];
let myDictionary: StringDictionary = { fila: '5', columna: 'C'}
```

Extensión Interfaces y Tipos de intersección

- Herencia de interfaces

```
interface Colorful {
  color: string;
}
interface Circle {
  radius: number;
}
interface ColorfulCircle extends Colorful, Circle {
  active: boolean;
}
const cc: ColorfulCircle = { color: "red", radius: 42, active: true };
```

- Tipos de intersección, similares a los interfaces pero con sutiles diferencias

```
type ColorfulCircle = Colorful & Circle;
const cc: ColorfulCircle = { color: "blue", radius: 42 }
```


Módulos

- A partir de ECMAScript 2015, JavaScript tiene un concepto de módulos. TypeScript contaba con su propio concepto de módulos y espacios de nombres (módulos internos) que adecuo para adoptar el propuesto por ECMAScript. También da soporte al cargador de Node.js para módulos CommonJS y el cargador RequireJS para módulos AMD en aplicaciones web.

```
1  module Sayings {
2      export class Greeter {
3          greeting: string;
4          constructor(message: string) {
5              this.greeting = message;
6          }
7          greet() {
8              return "Hello, " + this.greeting;
9          }
10     }
11 }
12 var greeter = new Sayings.Greeter("world");
```

Módulos (ES6)

- Los módulos se basan en el concepto de ficheros como módulos:
  - Cualquier archivo que contenga un nivel superior import o export se considera un módulo.
  - Solo se puede importar lo previamente exportado, lo no exportado es privado del módulo
  - Es necesario importar antes de utilizar
  - El fichero en el from sin extensión y ruta relativa (./ ../) o sin ruta (NODE_MODULES)
- Para exportar:

```
export public class MyClass { }
export default class MyDefaultClass {}
export { MY_CONST, myFunction, name as otherName }
```

- Para importar:

```
import * from './my_module';
import * as MyModule from './my_module';
import MyDefaultClass, { MyClass, MY_CONST, myFunction as func } from './my_module';
```

- Directorios como módulos (Agrupación, Reexportaciones, Barrel): Importar y exportar (index.ts)

```
export { MyClass, MY_CONST, myFunction as func } from './my_module';
```


Decoradores (ES7)

- Existen ciertos escenarios que requieren funciones adicionales para admitir la anotación o modificación de clases y miembros de clase. Los decoradores proporcionan una forma de agregar anotaciones y una sintaxis de metaprogramación para declaraciones de clase y miembros.
- Un decorador es un tipo especial de declaración que se puede adjuntar a una declaración de clase, método, descriptor de acceso, atributo o parámetro. Los decoradores usan el formato @expression, donde expresión se debe evaluar como una función que se llamará en tiempo de ejecución con los metadatos sobre la declaración decorada.

```
function anotacion(value: string) {
  // configura la función decorador devuelta
  return function (target: any, propertyKey: string, descriptor: PropertyDescriptor) {
    // este es el decorador, hace algo con 'target' (elemento anotado) y 'value'...
  };
}
@anotacion('valor')
abstract class Persona {
    @notBlank
    private nombre: string;
```

asyc/await (ES2017)

- La sintaxis asyc/await permite un estilo de programación secuencial en procesos asíncronos, delegando en el compilador o interprete su implementación.
- La declaración de función asyc define una función asíncrona, que devuelve un objeto AsyncFunction. Una función asíncrona es una función que opera asincrónicamente a través del bucle de eventos, utilizando una promesa implícita para devolver su resultado. Pero la sintaxis y la estructura de su código usando funciones asíncronas se parece mucho más a las funciones síncronas estándar.
- El operador await se usa para esperar a una Promise y sólo dentro de una async function.

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => { setTimeout(() => { resolve(x); }, 2000); });
}
async function f1() {
  let x = await resolveAfter2Seconds(10);
  console.log(x); // 10
}
```


Compatibilidad con Frameworks JS

- Typescript permite comunicarse con código JavaScript ya creado e incluso añadirle “tipos” a través de ficheros de definiciones .d.ts que prototipan los tipos que reciben y devuelven las funciones de una librería.
- Ya existen miles de ficheros de definiciones (.d.ts) que incluyen la practica totalidad de los framework más populares (http://definitelytyped.org/).
  - npm install --save-dev @types/jquery
- Definiciones: https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types

```
1  declare module Sayings {
2      class Greeter {
3          greeting: string;
4          constructor(message: string);
5          greet(): string;
6      }
7  }
8  declare var greeter: Sayings.Greeter;
9
```

DOCUMENTADOR

Documentación

- Hacer la documentación del código fuente puede llegar a ser muy tedioso, todos los programadores prefieren ir directo al grano, escribir su código y pasar de largo esta aburrida tarea. Por fortuna, actualmente existen un montón de herramientas para agilizar la documentación del código sin tener que dedicarle más tiempo del imprescindible.
- JSDoc es una sintaxis para agregar comentarios con documentación al código fuente de JavaScript.
- La sintaxis JSDoc es similar a la sintaxis de Javadoc, usado para documentar el código de Java, pero se ha especializado para trabajar con la sintaxis de JavaScript, es más dinámico y, por tanto único, ya que no es totalmente compatible con Javadoc. Sin embargo, como Javadoc, JSDoc permite al programador crear Doclets y Taglets que luego se pueden traducir en formatos como HTML o RTF.

```
/**
 * Create a dot.
 * @param {number} x - The x value.
 * @param {number} y - The y value.
 * @param {number} width - The width of the dot, in pixels.
 */
constructor(x, y, width) {
```

JSDoc

Etiqueta	Descripción
@author	nombre del autor.
@constructor	indica el constructor.
@deprecated	indica que ese método es deprecated.
@exception	sinónimo de @throws.
@param	parámetros de documentos y métodos.
@private	indica que el método es privado.
@return	indica que devuelve el método.
@see	Indica la asociación con otro objeto.
@throws	Indica la excepción que puede lanzar un método.
@version	indica el número de versión o librería.

Documentador

- JDOC
  - http://usejsdoc.org/index.html
  - https://github.com/jsdoc3/jsdoc
  - http://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=881:guia-de-estilo-javascript-comentarios-proyectos-jsdoc-param-return-extends-ejemplos-cu01192e&catid=78&Itemid=206
- Doxygen
  - http://www.stack.nl/~dimitri/doxygen/
- Compodoc
  - https://compodoc.github.io/website/
- Typedoc
  - https://typedoc.org

JSX

JSX

- Es una extensión de sintaxis de JavaScript, creada por Facebook para el uso con su librería React, que permite incrustar XML (HTML) dentro de JavaScript sin tener que delimitarlo como cadenas. Invierte el concepto de scripting: HTML con JavaScript incrustado → JavaScript con HTML incrustado.

```
const element = (
  <h1 class='greeting'>
    Hola Mundo!!!
  </h1>
);
```

- JSX puede recordar un lenguaje de plantilla, pero cuenta con todo el poder de JavaScript.

JSX

- Stencil recomienda utilizar JSX para crear plantillas en lugar de JavaScript regular. No es necesario utilizarlo, pero hay importantes ventajas en su utilización.
- Ventajas:
  - Es **más eficiente**, ya que se realiza una optimización al compilar código de JavaScript.
  - Es **mas seguro**, dado que la mayoría de los errores pueden ser capturados durante la compilación.
  - Es **más fácil** y **rápido** escribir plantillas si se está familiarizado con el lenguaje HTML.
- Desventajas:
  - Requiere compilación.

Elementos

- Cada elemento representa una etiqueta HTML que puede contener otras etiquetas.

```
const element = (
  <p>
      Hola <b>Mundo!!!</b>
  </p>
);
```

- Se puede dividir JSX en varias líneas para facilitar la lectura. Si bien no siempre es necesario, al hacer esto, también se recomienda envolverlo entre paréntesis para evitar los inconvenientes de la inserción automática de punto y coma. Debe estar correctamente anidado y todas las etiquetas deben cerrarse, las etiquetas vacías se cierran con /> (similar a XML):
  - …<input … />
…
- Por defecto solo puede contener una única etiqueta, para contener mas de una es necesario envolverlas en un contenedor, un array o un fragmento (sin nodo DOM), el fragmento acepta la notación diamante <>:

```
const element = (
  <div>
    <h1>Hola</h1>
    <h2>Mundo</h2>
  </div>
);
```

```
const element = [
  <h1>Hola</h1>
  <h2>Mundo</h2>
];
```

```
const element = (
  <Fragment>
    <th>Hola</th>
    <td>Mundo</td>
  </Fragment>
);
```

```
const element = (
  <>
    <h1>Hola</h1>
    <h2>Mundo</h2>
  </>
);
```

Expresiones

- Se puede incrustar cualquier expresión de JavaScript en un fragmento HTML envolviéndola con llaves.

```
const calculo= <p>Calcula: 2 + 2 = {2 + 2}</p>;
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}
const saludo= <h1>Hola {formatName(user)}</h1>;
```

- Para asignar valores a los atributos se utilizan las comillas para los valores constantes y las llaves para las expresiones (y constantes aritméticas), los objetos se deben usar con dobles llaves:

```
const constante = <img src="http://lorempixel.com/64/64/people/" />;
const expresion = <img src={user.avatarUrl} width={200} />;
const error = <output style={{ color: 'red' }}>{msg}</output>
```

- Los comentarios son expresiones que se ignoran:

```
{/* … */}
```


Expresiones complejas

- Si no se asigna valor al atributo se establece de manera predeterminada con el valor true.

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

- Si ya se dispone de las propiedades en un objeto se puede pasar utilizando el operador ... de "propagación" (ES2015):

```
const attr = {init: 10, delta: '1'};
return <Contador {...attr} />; // <Contador init={10} delta="1" />
```

- El operador de "propagación" también permite la des-estructuración:

```
const { tipo, ...attr } = props;
if (tipo)
   return <Contador {...attr} />;
return <Slider {...attr} />;
```

Expresiones condicionales

- Operador condicional simple:
  ```
  const saludo= <p>Hola {user && <b>formatName(user)</b>}</p>;
  ```
- Operador condicional ternario:
  ```
  const btnLogIn = <input type="button" value="Log In" />;
  const btnLogOut = <input type="button" value="Log Out" />;
  const saludo= <p>{user ? btnLogIn : btnLogOut}</p>;
  ```
- Si se encapsula en una función o expresión lamda se pueden utilizar instrucciones condicionales y bucles:
  ```
  function saludar(user) {
    if (user) {
      return <h1>Hola {formatName(user)}!</h1>;
    }
    return <h1>Hola desconocido.</h1>;
  }
  ```
- No se renderizan los valores true, false, null y undefined.
  `<div>{true}</div>` *se visualiza como* `<div></div>`

Expresiones iterativas

- Se pueden crear colecciones de elementos e incluirlos en JSX con llaves {}.
  ```
  const numbers = [1, 2, 3, 4, 5];
  const listItems = numbers.map((number) =>  <li key={number.toString()}>{number}</li> );
  const list = <ul>{listItems}</ul>;
  const otra = <ul>{numbers.map((number) => <li key={number.toString()}>{number}</li> )}</ul>;
  ```
- ***key*** es un atributo cadena especial que debe incluir al crear listas de elementos.
- Las claves ayudan al VirtualDOM a identificar qué elementos han cambiado, se han agregado o se han eliminado. Las claves se deben asignar dentro de la matriz para dar a los elementos una identidad estable. La mejor forma de elegir una clave es usar una cadena que identifique de forma única cada elemento. Muy a menudo se utiliza como claves la clave primaria del modelo de datos.
- Cuando no hay IDs estables para los elementos representados y como último recurso, se puede usar el índice en el array como clave (no se debe ordenar ni eliminar elementos):
  ```
  const listItems = numbers.map((number, indice) => <li key={indice.toString()}>{number}</li> );
  ```

XSS

- De forma predeterminada, Stencil "escapa" cualquier valor incrustado en JSX antes de representarlos garantizando que nunca se inyectará nada que no esté escrito explícitamente en la aplicación. Todo se convierte en una cadena antes de ser renderizado. Esto ayuda a prevenir ataques XSS (cross-site-scripting).
- En general, es arriesgado establecer contenido HTML desde el código sin el correspondiente saneamiento, porque se puede exponer inadvertidamente a los usuarios a un ataque cross-site scripting (XSS).
- Por lo tanto, para establecer contenido HTML directamente desde Stencil, se debe usar la propiedad innerHTML y pasarle el valor previamente saneado.

```
<div innerHTML={svgContent}></div>
```


COMPONENTES

Composición

- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente. Sigue un modelo de composición jerárquico con forma de árbol de componentes. La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Un botón, un formulario, un listado, una página: todos ellos se expresan comúnmente como componentes.
- El API de Componentes de Stencil proporciona un conjunto de decoradores, hooks del ciclo de vida y métodos de representación para la creación de componentes, la mayoría de ellos se importan del paquete @stencil/core.
- Los decoradores o anotaciones son una construcción en tiempo de compilación pura utilizada por Stencil para recopilar todos los metadatos sobre un componente, las propiedades, los atributos y los métodos que podría exponer, los eventos que podría emitir o incluso las hojas de estilo asociadas. Una vez que se han recopilado todos los metadatos, todos los decoradores se eliminan de la salida, por lo que no incurren en ninguna sobrecarga de tiempo de ejecución.

Componente

- Un componente es una clase decorada con la anotación @Component() que debe proporcionar en la propiedad tag un nombre HTML para el componente. A menudo, también se usan las propiedades styleUrl para proporcionar una hoja de estilo.

```
$ npm init stencil component my-contador
```

```
@Component({
  tag: 'my-contador',
  styleUrl: 'my-contador.css',
  shadow: true,
})
export class Contador {
   render() { return <h1>Hola mundo</h1> }
}
```

- A la hora de estructurar los componentes se recomienda: Un componente por archivo y un componente (la implementación .tsx y sus estilos) por directorio, aunque puede tener sentido agrupar componentes similares en el mismo directorio, es más fácil documentar los componentes cuando cada uno tiene su propio directorio.

Nomenclatura

- El prefijo en la etiqueta tiene un papel importante cuando crea una colección de componentes destinados a ser utilizados en diferentes proyectos. Los componentes web no tienen alcance porque se declaran globalmente dentro de la página web, lo que significa que se necesita un prefijo "único" para evitar colisiones, que también ayuda a identificar rápidamente la colección de la que forma parte. Además, la especificación requiere que contengan un guión "-" dentro del nombre de la etiqueta (kebab-case), usar la primera sección para distinguir sus componentes es lo natural.
- Los componentes son conceptualmente cosas, no son acciones, deben usar sustantivos en lugar de verbos en el nombre de etiqueta. Cuando varios componentes están relacionados y/o acoplados, es una buena idea compartir el nombre y luego agregar diferentes modificadores.
- Los nombres de la clase del componente utilizan PascalCase y no necesitan tener un prefijo ya que las clases tienen alcance y no hay riesgo de colisión.

```
@Component({ tag: 'my-card', styleUrl: 'my-card.css', shadow: true, })
export class Card { }
@Component({ tag: 'my-card-header', styleUrl: 'my-card-header.css', shadow: true, })
export class CardHeader { }
@Component({ tag: 'my-card-content', styleUrl: 'my-card-content.css', shadow: true, })
export class CardContent { }
```


Definición del componente

- Un Web Component, como cualquier otro HTMLElement, puede exponer:
  - Propiedades (AtributosHTML y propiedades DOM) para permitir su personalización.
  - Métodos para permitir la interacción.
  - Eventos para notificar sus cambios
- Todos ellos, así como la clase, deben contar con su documentación JSDocs dado que forman parte del interfaz externo del Web Component.
- La clase del componente, además de la implementación de los miembros propios del Web Component, debe contar siempre con un método render(), que utilice TSX/JSX, responsable de la representación del componente.
- Adicionalmente, puede contar con:
  - Atributos o campos de la clase (sin renderizado).
  - Estado, activa el renderizado.
  - Métodos de la clase y controladores de eventos.

Renderizar el componente

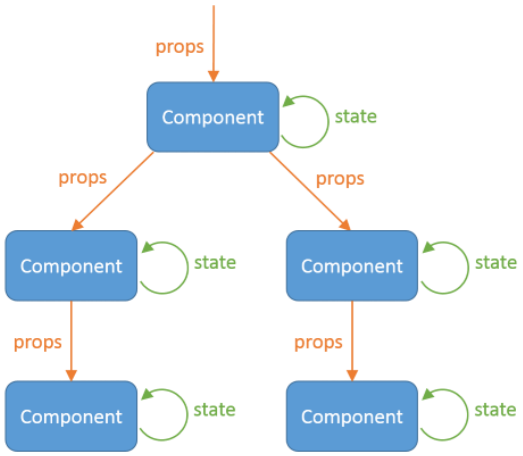
- La clase del componente debe contar siempre con un método render() que devuelve el TSX/JSX responsable de la representación del componente o null si no se representa. El método puede utilizar al resto de miembros de la clase para generar el resultado, pero no puede provocar cambios que impliquen el repintado.
- Dentro del TSX se pueden utilizar etiquetas de HTML y de componentes, interactuando siempre de la misma forma: atributos HTML/propiedades DOM/eventos. La utilización de etiquetas de componentes dentro del diseño de los componentes crea el modelo de composición jerárquico con forma de árbol de componentes con un mínimo acoplamiento.
- Los elementos de Stencil son inmutables. Una vez se crea un elemento, no puede cambiar sus elementos secundarios o atributos. Un elemento es como un único fotograma en una película: representa la UI en un momento determinado. La única forma de actualizar la UI es crear un nuevo elemento y volver a ejecutar render(). Stencil supervisa las propiedades y el estado, cuando detecta cambios vuelve a ejecutar el método render(): activa el renderizado.
- Stencil compara el elemento y sus elementos secundarios con el anterior (VirtualDOM), y solo aplica las actualizaciones DOM necesarias para llevar el DOM al nuevo estado. La filosofía de Stencil es pensar en cómo debe verse la UI en un momento dado en lugar de cómo cambiarla a lo largo del tiempo, esto elimina muchos tipos de errores.

Renderizar el componente

```
render() {
  return (
    <Host class={{ 'negativo': this.esNegativo()}} >
      <div class="center">
        <my-pantalla display={this.counter} />
        <div class="buttons">
          <button type="button" class="button-left" onClick={() => this.baja()}>-</button>
          <button type="button" class="button-right" onClick={() => this.sube()}>+</button>
        </div>
      </div>
    </Host>
  );
}
```

Flujo de datos

- La información fluye hacia abajo en el árbol de componentes.
- Los componentes, contenedores o contenidos, no pueden saber si un componente determinado tiene o no estado.
- Esta es la razón por la cual a menudo se denomina al estado como local o encapsulado: no es accesible por ningún componente que no sea el que lo posee y establece.
- Un componente puede elegir pasar su estado como propiedad a sus componentes contenidos.
- Esto se conoce comúnmente como flujo de datos "descendente" o "unidireccional". Cualquier estado siempre es propiedad de algún componente específico, y cualquier dato o IU derivado de ese estado solo puede afectar a los componentes por "debajo" de ellos en el árbol de componentes.
- Se denomina elevar estado al mecanismo mediante el cual un componente comunica sus cambios de estado a su contenedor, habitualmente mediante los eventos.



Propiedades

- Los propiedades definen los atributos de la etiqueta HTML y las propiedades del DOM del componente. Los cambios en las propiedades activan el renderizado del componente.
- La propiedades se declaran en línea como atributos de la clase anotados con @Prop(), con su correspondiente documentación dado que forman parte del interfaz externo del Web Component:

```
export class Contador {
  /** Valor inicial del contador */
  @Prop() initValue: number = 0;
  /** Valor a incrementar/decrementar del contador */
  @Prop() delta = 1;
```

- En el ecosistema de JavaScript, es común usar camelCase al nombrar propiedades, pero en HTML, que no es sensible a la tipografía, se debe usar dash-case (kebab-case). Stencil lo resuelve automáticamente:

```
<my-contador initValue={10}></my-contador> {/* en TSX o JSX */}
<my-contador init-value="10"></my-contador> <!-- en HTML -->
```

Atributo HTML vs Propiedad DOM

- El atributo HTML (tiempo de diseño) y la propiedad DOM (tiempo de ejecución) nunca son lo mismo, incluso cuando tienen el mismo nombre. Los valores de una propiedad pueden cambiar, mientras que el valor del atributo no puede cambiar una vez expresado (forma en la que está escrito el documento HTML). Los atributos pueden inicializar propiedades, siendo las propiedades las que posteriormente pueden cambiar.
- Las propiedades y los atributos de los componentes están fuertemente conectados, pero no necesariamente son lo mismo. En Stencil, el decorador @Prop() aplicado a una propiedad indicará al compilador de Stencil que genere el atributo HTML, la propiedad DOM y que escuche los cambios de la propiedad en DOM. Por lo general, el nombre de una propiedad es el mismo que el atributo, pero no siempre es así:
  - Solo se genera el atributo HTML si es un tipo primitivo (number, boolean o string)
  - Las propiedades DOM siguen la notación camelCase y los atributos HTML usan dash-case
- Este comportamiento se puede cambiar con la opción attribute del @Prop():

```
@Prop({ attribute: 'value' }) initValue: number = 0;
@Prop({ attribute: 'options' }) optionsService: OptionsService;
<my-contador value="10" options="{ update: false }"></my-contador> <!-- en HTML -->
```


Tipado de Propiedades

- Las propiedades pueden ser boolean, number, string, object o array (los tipos JavaScript).
- Los booleanos se inicializan con {true} y {false} en TSX. En HTML se inicializan con "true", "false", sin declarar ("false") o declarado sin asignar valor ("true").
- Los numéricos utilizan {valor} en TSX y cadenas en HTML que automáticamente se convierten a su valor numérico o a NaN si no es posible.
- Las propiedades con otros tipos TypeScript se validan en TSX, pero no generan atributos HTML, solo propiedades DOM de tipo object que solo se pueden manejar mediante JavaScript y no se validan los tipos. Lo mismo pasa con las propiedades de tipo array.
- TypeScript permite que las propiedades se marquen como:
  - opcionales, añadiendo una ? al final del nombre del miembro (undefined si no se asigna valor).
  - con valor predeterminado, a usar si no se asigna valor en vez del undefined predeterminado.
  - requeridas, añadiendo una ! al final del nombre del miembro.
  - any, pueden contener cualquier valor, par evitar la verificación de tipos de un valor específico.
  - union, lista de posibles tipos.
- Estas validaciones solo se realizan en TSX.

Mutabilidad de Propiedades

- Una propiedad es inmutable por defecto desde dentro de la lógica del componente. Su valor debe establecerse externamente, el componente no debe actualizarlo internamente o se generara un warning. Los componentes deben comportarse como “funciones puras“ porque no deben cambiar sus entradas y siempre devuelven el mismo resultado para las mismas entradas.
- Es posible permitir explícitamente que un Prop se mute desde dentro del componente aunque no recomendable, declarándolo como mutable:

  ```
  @Prop({ attribute: 'value', mutable: true }) initValue: number = 0;
  ```
- Stencil compara Props por referencia para volver a renderizar componentes de manera eficiente. La configuración mutable: true en una propiedad object o array permite que su referencia cambie dentro del componente y active el renderizado. Por razones de rendimiento solo mira la referencia por lo que no detecta los cambios en el object o array referenciado. En su lugar, se debe sustituir la referencia con un nuevo objeto mutado que active el renderizado.

Reflejar valores de propiedades

- En algunos casos, puede ser útil reflejar los valores deuna propiedad sincronizada como atributos de la etiqueta HTML para mantener con un atributo. En este caso, puede establecer la opción { reflect: true } en el decorador @Prop(). Cuando se refleja una propiedad, se representará en el DOM como un atributo HTML.
- Cuando en el JSX/TSX del render del componente se utiliza:

  ```
  <my-contador initValue={this.init}></my-contador>
  ```
- Se muestra en HTML:

  ```
  <my-contador></my-contador>
  ```
- Si se activa el reflejo en las propiedades deseadas:

  ```
  export class Contador {
    /** Valor inicial del contador */
    @Prop({ reflect: true }) initValue: number = 0;
    /** Valor a incrementar/decrementar del contador */
    @Prop({ reflect: true }) delta = 1;
  ```
- Ahora se mostraría en HTML:

  ```
  <my-contador delta="1" init-value="10"></my-contador>
  ```

Slots

- Los componentes son etiquetas que pueden tener contenido, por lo que a menudo se necesitan representar elementos secundarios dinámicos en ubicaciones específicas en su árbol de componentes, lo que permite que un desarrollador proporcione contenido secundario cuando usa nuestro componente, con nuestro componente colocando ese componente secundario en la ubicación adecuada. Para hacer esto, se usa la etiqueta Slot dentro del render() para indicar donde mostrar el contenido recibido.

```
render() {
  return (
    <div>
      <h2>{this.title}</h2>
      <slot />
    </div>
  );
}
<my-card title="Saludo"><p>Hola mundo</p></my-card>
```


Múltiples Slots

- En algunos casos el contenido se debe repartir en varias ubicaciones dentro del componente. El componente puede usar mas de una etiqueta Slot si cuentan en el atributo name con un identificador único.
- Para hacer esto, se usa dentro del render() para indicar donde mostrar el contenido recibido.

```
render() {
  return (
    <div>
      <slot name="title" />
      <slot name="body" />
    </div>
  );
}
```

- El contenido, a través del atributo slot indica el identificador del slot de destino:

```
<my-card><h1 slot="title">Saludo</h1><p slot="body">Hola mundo</p></my-card>
```

Cambios en el slot

- Se pueden detectar los cambios es el slot:

```
child!: HTMLElement;
handleSlot(ev){
   console.log('Slot', ev)
 }
<slot onSlotchange={ev => this.handleSlot(ev)} ref={el => this.child = el as HTMLElement} />
```

- El contenido del slot no pertenece al componente, pertenece a uno de sus contenedores, por lo que las acciones sobre el slot están sumamente limitadas.

Estado

- El estado es un término general que se refiere a los valores y objetos que se almacenan en una clase o una instancia de una clase para su uso ahora o en el futuro.
- Al igual que una clase de TypeScript normal, un componente Stencil puede tener uno o más atributos o campos para contener valores que conforman el estado del componente. Stencil permite marcar con el decorador @State() aquellos cullos cambios deben activar un nuevo renderizado del componente.

```
export class Contador {
  @State() private counter = 0;
```

- El estado debe ser privado y solo se puede cambiar desde dentro de la propia clase. Solo debe usarse cuando sea absolutamente necesario en aquellos que al cambiar deban activar el renderizado porque cambian la representación del componente: tienen efectos visibles en el interfaz de usuario.

Tipos complejos en el Estado

- Cuando Stencil verifica si un atributo o campo con @Prop() o @State() ha cambiado la referencia ha cambiado para detectar los cambios, pero si es un objeto o un array y cambia su contenido sin cambiar la referencia y no se detectara el cambio.
- Las operaciones mutables estándar sobre arrays, como push() y unshift() no se detectan. Para realizar cambios en un array, se deben utilizar operadores no mutables, como el operador de propagación map() y filter(), que devuelven un nuevo array que debe asignarse al atributo que se está observando, al ser una nueva referencia se puede detectar el cambio.

```
@State() items: Item[] = [];
this.items = [...this.items, newItem];
this.items.splice(1)
this.items = [...this.items];
```

- En los objetos, al igual que con los arrays, hay que crear una copia y sobre la copia se realizan las mutaciones, cuando se sustituye el original por la copia se detectaran los cambios. La copia se puede realizar con el operador de propagación o con Object.assign().

```
@State() item: Item;
this.item = Object.assign({}, this.item , {description: newDescription})
this.item = { ...this.item, description: newDescription}
```


Vigilar propiedades y estado

- La anotación @Watch() se aplica al método de un componente Stencil que se ejecutará automáticamente cuando cambie su miembro de clase asociado. El método recibirá los valores antiguos y nuevos de prop/state, lo que es útil para la validación o el manejo de efectos secundarios. El decorador acepta un solo argumento, el nombre del atributo o campo @Prop() o @State() a vigilar. El @Watch() no se dispara cuando un componente se carga inicialmente.

```
@Watch('init')
initChanged(newValue: number, _oldValue: number) {
  if (this.counter !== newValue) { this.counter = newValue; }
}
@Watch('delta')
validateDelta(newValue: number, oldValue?: number) {
  if (newValue < 1) {
    this.delta = oldValue
    throw new Error('No puede tener un delta negativo.')
  }
}
componentWillLoad() {
  this.initChanged(this.init, this.counter)
  this.validateDelta(this.delta, 1)
}
```

Métodos

- Los elementos del DOM exponen métodos para interactuar con ellos desde JavaScript. De igual forma, los Web Component pueden exponer métodos destinados a ser invocados desde el exterior.
- La arquitectura de Stencil es asíncrona en todos los niveles, lo que permite muchos beneficios de rendimiento y facilidad de uso. Los métodos expuestos públicamente deben devolver una promesa:
  - Los desarrolladores pueden llamar a los métodos antes de que se descargue la implementación sin componentOnReady(), se ponen en cola las llamadas al método y las resuelve cuando el componente haya terminado de cargarse.
  - La interacción con el componente es la misma ya sea que utilice una carga lenta o que ya esté completamente hidratado.
  - Al mantener asíncrona la API pública de un componente, las aplicaciones podrían mover los componentes de forma transparente a Web Worker y la API seguiría siendo la misma.
  - Solo se requiere que los métodos expuestos públicamente devuelvan una promesa, los demás métodos de componentes son privados para el componente y no es necesario que sean asincrónicos.

Métodos

- Se usa la anotación @Method() para decorar el método, con nombre en camelCase, que se debe exponer en la API pública. Como parte de la API pública se deben documentar.
- Deben devolver una promesa, pero puede estar ya resuelta:

```
@Method()
getContador() { return Promise.resolve(this.counter); }
```

- Los métodos async ya devuelven una promesa:

```
@Method()
async getContador() { return this.counter; }
```

- Incluso si no devuelve nada, debe ser asíncrono

```
/** Reinicia el contador */
@Method()
async reset() {
 this.counter = 0;
}
```

Eventos

- Stencil proporciona una API para especificar los eventos DOM que un componente puede emitir y los eventos DOM que un componente escucha, no cuenta con un sistema eventos propio.
- Los componentes pueden emitir datos y eventos:
  - se declaran como atributos del tipo EventEmitter<TipoEmitido> decorados con la anotación @Event()
  - con la documentación para interfaz del Web Component
  - se inicializan internamente por lo que no deben inicializarse manualmente.
  - pueden emitir valores, en cuyo caso se encapsulan en un único valor de TipoEmitido que se puede consultar en la propiedad detail del evento sintético (CustomEvent)
  - para emitir el evento y sus datos se utiliza el método emit() del EventEmitter.

```
/**
 * Notifica los cambios en el contador
 */
@Event() updated: EventEmitter<number>;

protected onUpdated() {
  this.updated.emit(this.counter)
}
```


Controlador de Eventos

- En HTML:

```
document.querySelector('my-contador').addEventListener('updated', event => console.log(event.detail))
```

- En TSX/JSX: los eventos se manejan de una forma muy similar a como se manejan los eventos en HTML/DOM, pero con algunas diferencias importantes:
  - Los eventos se vinculan con el prefijo on en notación camelCase, tanto personalizado como del DOM.
  - En la expresión se pasa una función/método (callback) como controlador de eventos: el objeto, no la invocación.
  - Se pueden utilizar expresiones lamda, pero tienen la particularidad de crear un nuevo objeto función de devolución de llamada cada vez que se renderiza.

```
handleEvent(ev: CustomEvent<number>) {
  console.log(ev.detail)
}
<my-contador onUpdated={this.handleEvent} />
<my-contador onUpdated={ev => this.handleEvent(ev)} />
<my-contador onUpdated={ev => console.log(ev.detail)} />
```

Objeto evento

- Por defecto el controlador de eventos recibe como argumento un objeto evento sintético (SyntheticEvent), alrededor del evento nativo del navegador.
- Para pasar argumentos adicionales a los controladores, cuando la expresión es un callback, se utiliza el método bind() de las funciones, con el objeto this como primer parámetro. El objeto evento se inyectara automáticamente como último argumento.

```
deleteRow(id: number, event: Event) { … }
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

- Se debe llamar a ev.preventDefault() explícitamente, devolver false no evita el comportamiento predeterminado (propagación de eventos). El objeto Event se reutilizará y todas las propiedades se anularán después de que se haya invocado el controlador de eventos, hay que cachear su información para acceder al evento de manera asincrónica.
- El parámetros puede ser del tipo base Event, CustomEvent para los emitidos o los específicos: KeyboardEvent, InputEvent, MouseEvent, WheelEvent, TouchEvent, DragEvent, FocusEvent, AnimationEvent, ClipboardEvent, HashChangeEvent, PageTransitionEvent, PopStateEvent, ProgressEvent, StorageEvent, TransitionEvent, UiEvent

Opciones de los Eventos

- El decorador @Event(opts: EventOptions) acepta opcionalmente un objeto de opciones para dar forma al comportamiento de los eventos enviados, con las siguiente propiedades
  - eventName: permite dar un nombre personalizado el evento.
  - cancelable: indica si el evento se puede cancelar (acepta event.preventDefault()).
  - bubbles: que indica si el evento se propaga hacia los contenedores.
  - composed: indica si puede propagarse a través del límite del shadow DOM hacia el DOM normal.

```
@Event({
  eventName: 'valueChange', cancelable: true, bubbles: true, composed: true,
}) updated: EventEmitter<number>;
protected onUpdated() {
  const event = this.updated.emit(this.counter)
  if(!event.defaultPrevented) {
    // Cancelar
  }
}
```

Detectores Eventos

- El decorador @Listen() permite al componente escuchar eventos DOM, incluidos los enviados desde @Events. Los detectores de eventos se agregan y eliminan automáticamente cuando el componente se agrega o elimina del DOM.
- La anotación @Listen() recibe el nombre del evento a capturar y decora el método o controlador de eventos a ejecutar cuando se produce la notificación.

```
@Listen('updated')
updatedHandler(event:CustomEvent<number>) {
 console.log('Listen: ', event.detail);
}
@Listen('keydown')
handleKeyUpDown(ev: KeyboardEvent){
 switch(ev.key) {
   case 'ArrowUp': this.sube(); break;
   case 'ArrowDown': this.baja(); break;
 }
}
```


Opciones de los Detectores Eventos

- Los controladores también pueden detectar un evento que no sea propio. La opción target de @Listen() se puede usar para definir dónde se adjunta el detector de eventos: 'body' | 'document' | 'window'.

```
@Listen('scroll', { target: 'window' })
handleScroll(ev) { console.log('the body was scrolled', ev); }
```

- Una forma alternativa de propagación de eventos (burbujeo) es la captura de eventos, pero en orden inverso: el evento se activa primero en el elemento menos anidado y luego en más elementos anidados sucesivamente, hasta que se alcance el objetivo. De forma predeterminada, @Listen() actúa en la fase de propagación pero con la opción { capture: true } actuara en la fase de captura.

```
@Listen('click', { capture: true })
handleClick(ev: Event) {
   console.log(`Clicked on ${(ev.currentTarget as HTMLElement).tagName}`);
}
```

@Element()

- La anotación @Element() permite obtener acceso al elemento host dentro de la instancia de clase. Esto devuelve una instancia de un HTMLElement, por lo que se pueden usar los métodos/eventos DOM estándar.

```
export class TodoList {
  @Element() el: HTMLElement;

  getListHeight(): number {
    return this.el.getBoundingClientRect().height;
  }
  :
}
```

- Si se necesita actualizar el elemento host en respuesta a cambios de estado o propiedades, se debe hacer en el método render() que usa el elemento <Host>.

Campos Referencia

- En el flujo de datos típico de React, las propiedades son la única forma en que los componentes principales interactúan con sus hijos. Para modificar un hijo, se vuelve a representar con nuevas propiedades. Sin embargo, hay algunos casos en los que se necesita modificar imperativamente a un hijo o ejecutar algunos de sus métodos, que podría ser un elemento DOM o un componente, fuera del flujo de datos típico.
- Para estos casos Stencil proporciona una vía de escape: ref. Mediante una función permite asociar un elemento DOM o componente a un atributo o campo de la clase:

```
btnSalir!: HTMLButtonElement;
<button type='button' ref={tag => this.btnSalir = tag as HTMLButtonElement}>Salir</button>
```

- Posteriormente, una vez ejecutado el render() apropiado, se puede acceder a el:

```
this.btnSalir.focus();
```

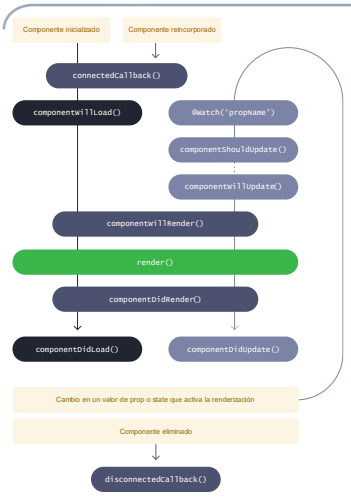
- Hay que evitar usar refs para cualquier cosa que se pueda hacer directamente de manera declarativa, aunque algunos buenos escenarios para usarlas son:
  - Administrar el foco y la selección de texto o la reproducción de medios.
  - Controlar animaciones.
  - Integración con bibliotecas DOM de terceros.

Manejo de la entrada del usuario

- Los elementos de formulario HTML funcionan de forma un poco diferente del resto de elementos DOM porque mantienen un estado interno distinto del estado de la clase.
- Para mantener sincronizados ambos estados, el componente debe controlar, en dichos elementos, lo que sucede después de la entrada del usuario mediante el uso de eventos.

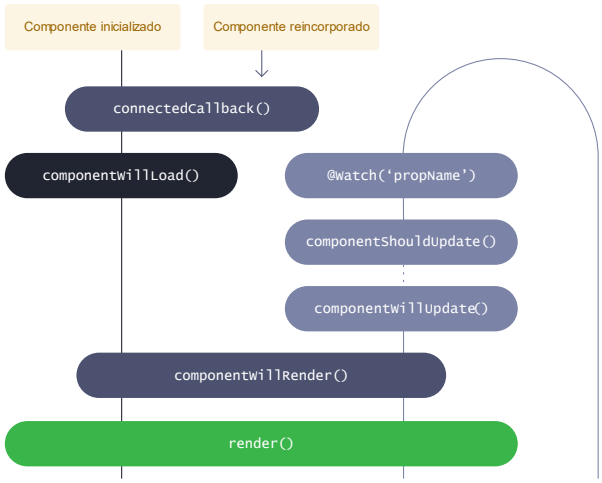
```
@State() nombre: string = 'Mundo';
@State() errors: {[index: string]: string } = {};
handleChange(event) {
  let cntr = event.target as HTMLInputElement
  this.nombre = cntr.value;
  this.errors[cntr.name] = cntr.validationMessage || null;
  this.errors = {...this.errors}
}
<h1>Hola {this.nombre}</h1>
<label for="nombre"> Nombre: </label>
<input type="text" id="nombre" name="nombre" value={this.nombre} required minLength={2}
        onInput={(event) => this.handleChange(event)} />
<output style={{color: 'red'}} hidden={!this.errors?.nombre}>ERROR: {this.errors?.nombre}</output>
```


Ciclo de vida del componente



- Para comprender el comportamiento de los componentes es importante comprender su ciclo de vida y los métodos que intervienen en dicho ciclo.
- Los componentes tienen numerosos métodos (callback o hook) de ciclo de vida que se pueden usar para saber cuándo se cargará, actualizará y renderizará el componente. Estos métodos se pueden agregar a un componente para conectarse a las operaciones en el momento adecuado. Cada método tiene un prefijo will o did dependiendo de si ocurren antes o después del render.
- Podemos plantear tres fases que pueden darse en el ciclo de vida de un componente.
  - Montaje o inicialización (única)
  - Actualización de propiedades o estado (múltiple)
  - Desmontaje (única)
- Si se implementa uno o varios de dichos métodos dentro de la clase del componente, Stencil los llamará automáticamente en el orden correcto.

Pre render del componente



- **connectedCallback**(): Llamado cada vez que el componente se conecta al DOM. Se llama antes componentWillLoad y se puede llamar más de una vez, cada vez que el elemento se adjunta o mueve en el DOM.
- **componentWillLoad**(): Llamado una vez justo después de que el componente se conecte por primera vez al DOM. Es un buen lugar inicializar el componente.
- **componentShouldUpdate**(newValue, oldValue, propName): boolean: Debe indicar si el componente debe volver a renderizarse o no.
- **componentWillUpdate**(): Solo se llama cuando se vuelve renderizar.
- **componentWillRender**(): Llamado siempre justo antes de renderizarse.

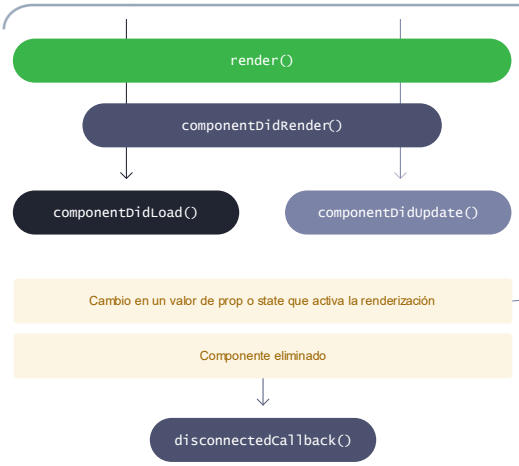
componentShouldUpdate

- Se llama cuando cambia una @Prop o un @State, después solicitarse un nuevo renderizado, para indicar que el proceso de renderizado no debe continuar y evitar renderizados innecesarios.
- Es el único que recibe argumentos: el valor nuevo, el valor antiguo y el nombre del @State o @Prop modificado. Debe devolver un valor booleano para indicar si el componente debe volver a renderizarse (true), sin consultar mas, o no (false).

```
componentShouldUpdate?(_newVal: any, _oldVal: any, propName: string) {
  if(propName === 'delta') return false;
  return true;
}
```

- Hay que tener método que, por ciclo de renderizado, se puede invocar tantas veces como @Prop o @State hayan cambiado. Cuando una devuelve true, indicando que el cambio tiene representación visual, el enlace no se volverá a llamar porque el renderizado ya está programada para suceder.
- El método debe ser una función pura, no realizar cambios, solo indicar si proceso debe continuar. Dado que no se ejecuta cuando ya está programado el renderizado, no es bueno confiar en él para observar los cambios de @Prop, en su lugar se debería utilizar un método decorado con @Watch.
- Este método no se ejecutará antes del renderizado inicial, si no se quiere generar contenido condicionalmente en la carga se debe implementar en render() para que devuelva null.

Post render del componente



- **componentDidRender**(): Llamado siempre justo después del renderizado.
- **componentDidLoad**(): Llamado solo una vez cuando el componente se ha renderizado por primera vez.
- **componentDidUpdate**(): Llamado cada vez que el componente se haya vuelto a renderizar.
- **disconnectedCallback**(): Se llama cada vez que el componente se desconecta del DOM, es decir, puede enviarse más de una vez.

Cambios de estado durante el ciclo de vida

- Se recomienda no cambiar el estado en los métodos del ciclo de vida, salvo que dependa estrictamente de ellos, para evitar renderizados adicionales que no es ideal para el rendimiento.
- En caso de cambiar el estado, siempre se recomienda realizar cualquier actualización de estado dentro componentWillRender() para cambios generales o en componentWillLoad() y componentWillUpdate() para los particulares. Se puede devolver una promesa para esperar al próximo renderizado.
- Si el estado se actualiza en componentDidRender(), componentDidLoad() o componentDidUpdate() activara otro renderizado y tiene el potencial de hacer que los componentes se atasquen en un bucle infinito. Si la actualización es inevitable, entonces el método también debe incluir una forma de detectar si las @Prop o el estado están "sucios" o no (los datos son realmente diferentes o son los mismos que antes). Al hacer una verificación de suciedad, se puede evitar renderizar los mismos datos y que se vuelva a llamar.

Métodos asíncronos de ciclo de vida

- Los métodos de ciclo de vida también pueden devolver promesas que permiten que el método recupere datos de forma asíncrona o realice cualquier tarea asíncrona.
- Un caso típico de esto es obtener datos de contenido antes de representarlos en un componente, pero las peticiones para cargar los datos remotos son lentas. Pero debido a que fetch() es asíncrono, es importante que componentWillLoad() devuelva un Promise para garantizar que su componente principal no se considere "cargado" hasta que todo su contenido se haya procesado.

```
componentWillLoad() {
  return fetch('/api/some-data')
    .then(response => response.json())
    .then(data => { this.content = data; });
}
```

- Los métodos componentWillLoad, componentWillRender y componentWillUpdate pueden devolver promesas.

Jerarquía del ciclo de vida

- Una característica útil de los métodos de ciclo de vida es que también tienen en cuenta el ciclo de vida de sus componentes secundarios, no se considera "cargado" hasta que hayan terminado de cargarse.
- También es importante tener en cuenta que, aunque Stencil puede cargar componentes de forma diferida y tiene un renderizado asíncrono, aún así los métodos del ciclo de vida se llaman en el orden correcto. Entonces, aunque el componente de nivel superior ya se haya cargado, esperará a que los componentes secundarios terminen de cargarse, así todos los métodos del ciclo de vida se llamaran en el orden correcto. Ocurre exactamente lo mismo al contrario, donde los componentes secundarios pueden ya estar listos mientras que el principal no lo está.
- Incluso si algunos componentes pueden o no estar ya cargados, toda la jerarquía de componentes espera a que sus componentes secundarios terminen de cargarse y renderizarse.

Desmontaje

- Le método disconnectedCallback() se llama cada vez que el componente se desconecta del DOM, es decir, puede enviarse más de una vez, no se debe confundir con algún tipo de evento "onDestroy" aunque en algunos escenarios se puede tratar como tales.
- Cuando el componente esta separado del árbol DOM, el navegador no lo representa, por lo que es un buen momento para realizar operaciones de limpieza como des suscribirse de los WebSocket o Stores, parar animaciones o ejecuciones de medias, cancelar peticiones HTTP que hayan quedado pendientes, eliminar listeners, temporizadores o demás objetos que puedan quedar en memoria.
- Es muy importante para evitar fugas de memoria o proceso, conservar los recursos y asegurar el rendimiento.

CSS

- La vinculación del CSS al componente se realiza a través de las propiedades de @Component:
  - styleUrl: URL relativa a un archivo de hoja de estilo externo. Debe ser un archivo .css a menos que se instale algún complemento externo como @stencil/sass.
  - styleUrls: Array similar a styleUrl pero que permite especificar diferentes hojas de estilo para diferentes modos.
  - styles: Cadena que contiene CSS en línea en lugar de usar fichero extorno, apropiado para pequeños fragmentos de CSS. Las características de rendimiento de esta función, una vez compilado, son las mismas que las de una hoja de estilo externa. Al no disponer de extensión solo se puede utilizar CSS, para usar SASS u otros es necesario utilizar styleUrl o styleUrls cuando se necesitan funciones más avanzadas.

Estilo

- En JSX/TSX, las propiedades style y class presentan algunas diferencias con el HTML/DOM.
- El propiedad style debe ser un objeto JavaScript plano, pares nombre/valor, donde: el nombre de la propiedad que debe coincidir con el nombre de la propiedad CSS, entrecomillado si es kebab-case (dispone de alternativas camelCase), y el valor entrecomillado con el valor CSS.

```
style={{'background-color': 'azure', fontWeight: 'bold', fontSize: '2em', opacity: "0.8"}}
```

- La propiedad class, recomenda, puede recibir una cadena o, simplifica el proceso cuando se manejan múltiples clases que aparecen y desaparecen, un objeto JavaScript plano, pares nombre/valor, donde: el nombre es el nombre de la clase CSS y el valor es un booleano que indica si el elemento tiene o no la clase.

```
class={{error: !this.isValid, urgente: this.isSpecial, importante: true }}
```

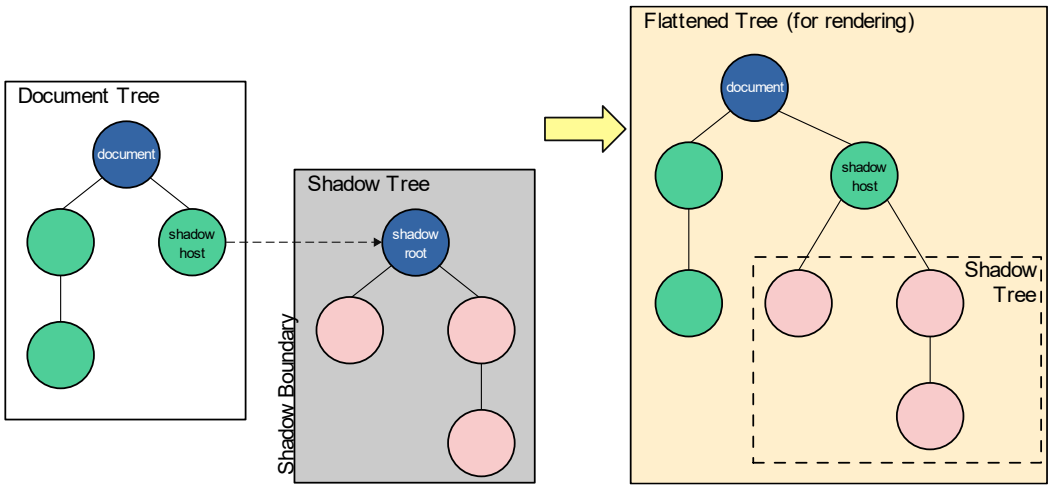
- Estos objeto pueden estar en un atributo o ser generados por métodos.

Shadow DOM

- Shadow DOM es una API integrada en el navegador que permite la encapsulación de DOM y la encapsulación de estilo. Es un aspecto central de los estándares de componentes web. El Shadow DOM protege los estilos, el marcado y el comportamiento de un componente del entorno que lo rodea. Esto significa que no tenemos que preocuparnos por aplicar nuestro CSS a nuestro componente, ni preocuparnos por que el DOM interno de un componente sea interferido por algo fuera del componente.
- Shadow DOM permite adjuntar arboles DOM en la sombra, ocultos a elementos en el árbol DOM regular. Este árbol Shadow DOM, shadow hosts, comienza con un elemento shadowRoot, debajo del cual se puede adjuntar cualquier elemento que desee, de la misma manera que el DOM normal.
- El Shadow DOM oculta y separa el DOM de un componente para evitar conflictos de estilos o efectos secundarios no deseados. Podemos usar el Shadow DOM en nuestros componentes Stencil para asegurarnos de que nuestros componentes no se vean afectados por las aplicaciones en las que se usan, agregando al opción shadow: true de @Component.

```
@Component({ tag: 'my-contador', styleUrl: 'my-contador.css', shadow: true, })
export class Contador {
```

Shadow DOM



CSS con Shadow DOM

- Cuando se usa el Shadow DOM y se desea consultar un elemento dentro de su componente web, debe usarse this.el.shadowRoot.querySelector(). Esto se debe a que todo el DOM dentro de su componente web está en un shadowRoot que crea el Shadow DOM.
- Con Shadow DOM habilitado, los elementos dentro del shadowRoot tienen scoped (alcance) y los estilos fuera del componente no se le aplican. Como resultado, los selectores de CSS dentro del componente se pueden simplificar, ya que solo se aplicarán a los elementos dentro del componente.
- Si un estilo en el DOM normal usa un selector que coincide con un elemento en el componente, esos estilos no se aplicarán.
- No tenemos que incluir ningún selector específico para aplicar estilos al componente.

CSS con Scoped

- Una alternativa al uso del shadow DOM es el uso de componentes con ámbito. Se puede usar componentes con ámbito configurando la opción scoped en @Component.

```
@Component({ tag: 'my-contador', styleUrl: 'my-contador.css', scoped: true, })
export class Contador {
```

- Scoped CSS es un proxy para la encapsulación de estilos. Funciona agregando un atributo de datos a sus estilos para que sean únicos y, por lo tanto, solo alcancen a su componente. Sin embargo, no evita que los estilos del DOM normal se filtren en su componente.
- Si es scoped: true, el componente utilizará la encapsulación nativa de shadow-dom, si el navegador es compatible, y recurrirá a scoped si no lo es . El valor predeterminado es falso.
- Si se proporciona un objeto literal que contiene `delegatesFocus`, el componente usará la encapsulación nativa shadow-dom. Cuando `delegatesFocus` se establece en `true`, el componente tendrá `delegatesFocus: true` agregado a su shadow DOM. Cuando `delegatesFocus` es `true` y se hace clic en una parte no enfocable del componente:
  - la primera parte enfocable del componente recibe el foco
  - el componente recibe cualquier estilo `focus` disponible
- Establecer `delegatesFocus` en `false` no agregará la propiedad `delegatesFocus` al shadow DOM y, por lo tanto, tendrá el comportamiento de enfoque descrito para `shadow: true`.

Propiedades personalizadas de CSS

- En CSS, las propiedades personalizadas (conocidas como variables) son entidades definidas por autores de CSS que contienen valores específicos que se pueden volver a utilizar en un documento.
- Se declaran en las reglas CSS con un nombre personalizado, sensible a la tipografía y prefijados por --.

```
:root {
    --error-color: red;
}
```

- Pueden tener un alcance global o local. El alcance es global si se declara dentro del pseudo selector :root. El alcance local permite su uso en el selector donde se declara o en cualquier selector anidado, primando las definiciones mas locales.
- La función var() permite recuperar el valor de la variable para definir el valor de otra propiedad. Opcionalmente se puede definir un valor por defecto para cuando no exista la variable.

```
.error-msg {
    color: var(--error-color, red);
    margin-top: calc(var(--gap) * 1px);
}
```

- Las variables CSS están en el DOM, por lo que se pueden cambiar con JavaScript.

Personalización de componentes con propiedades personalizadas

- Las propiedades personalizadas de CSS pueden permitir a los consumidores de un componente personalizar los estilos de un componente desde el DOM normal.
- Solo propiedades personalizadas de alcance es global, las declaradas dentro del pseudo selector :root, están expuestas a la aplicación consumidora.
- Si el componente define en su CSS:

```
:host {
  display: block;
  --negativo-back-color:  pink;
}
.negativo {
  background-color: var(--negativo-back-color);
}
```

- El consumidor puede definir en su hoja de estilos:

```
my-contador {
  --negativo-back-color: rgb(255, 69, 100);
}
```


CSS Shadow Parts

- La especificación CSS Shadow Parts define el pseudo-elemento ::part() y los atributos part y exportparts en los shadow hosts, lo que permite que los shadow hosts expongan selectivamente los elementos elegidos de su árbol oculto a la página exterior con el fin de poder aplicarles estilo.
- Cualquier elemento en un árbol oculto puede tener un atributo part, que se usa para exponer el elemento fuera del árbol de oculto. El atributo part contiene una lista separada por espacios con los nombres de la parte del elemento. Es mejor dar varios nombres a una parte, debe considerarse similar a una clase CSS, no una identificación o nombre de etiqueta.
- Cualquier elemento en un puede tener un atributo exportparts, pero si el elemento es un shadow hosts, permite exponer partes de elementos de su árbol oculto a las que se aplicaran las reglas CSS definidas fuera del shadow hosts (como si fueran elementos del mismo árbol).
- El pseudo-elemento ::part() permite seleccionar elementos a través del los nombres de partes, definidos con el atributo part, siempre que estén en el mismo árbol o hayan sido exportado con un atributo exportparts.

Personalización de componentes con Partes CSS

- Las propiedades personalizadas de CSS permiten personalizar valores individuales, por lo que tienen un alcance limitado. Para situaciones en las que el consumidor requiere una personalización mas compleja, las partes CSS ofrecen un mayor grado de flexibilidad en el estilo.
- Para definir partes del componente:

```
<output part="pantalla">{this.counter}</output>
```

- Para exponer sus partes:

```
<my-contador exportparts="pantalla"></my-contador>
```

- Para definir el estilo de una de las partes:

```
my-contador::part(pantalla) {
  font-size: 2em;
}
```


Integración con preprocesadores: SASS

- Este paquete se usa para precompilar fácilmente archivos Sass dentro de los componentes de Stencil. Internamente, utiliza una implementación JavaScript pura de Sass.
  - npm install @stencil/sass --save-dev
- Dentro de la configuración del proyecto Stencil, en stencil.config.ts, hay que importar el complemento y agregarlo a la propiedad plugins de la configuración:

```
import { Config } from '@stencil/core';
import { sass } from '@stencil/sass';

export const config: Config = {
  plugins: [
    sass()
  ]
};
```

- Este complemento precompilará automáticamente los ficheros que aparezcan en las styleUrl o stylesUrl con extensión .scss o .sass.

<Host>

- La etiqueta Host se puede utilizar en la raíz de la función de representación para establecer atributos y controladores de eventos en el propio elemento anfitrión. Esto funciona como cualquier otro JSX:

```
return (
    <Host aria-hidden={this.open ? 'false' : 'true'}
        class={{ 'todo-list': true, 'is-open': this.open }}
    />
  )
```

- <Host> también se puede utilizar cuando es necesario renderizar más de un componente en el nivel raíz de TSX, como un <Fragment>.
- Se puede definir en la hoja de estilos su estética, pero el selector cambia:
  - nombre-componente si no se utiliza Shadow DOM
  - el selector global :host si se utiliza Shadow DOM

Bundling

- Para que Stencil agrupe los componentes de manera más eficiente, se debe declarar un solo componente (clase decorada con @Component) por archivo de TypeScript, y el componente en sí debe tener un único export. Al hacerlo, Stencil puede analizar fácilmente todo el grafo de componentes dentro de la aplicación y comprender mejor cómo se deben agrupar los componentes. Debajo del capó, utiliza la utilida Rollup para agrupar de manera eficiente el código compartido. Además, la carga diferida es una función predeterminada de Stencil, por lo que la división de código ya se está realizando automáticamente y solo se importan dinámicamente los componentes que se utilizan en la página.
- Los módulos que contienen un componente son puntos de entrada, lo que significa que ningún otro módulo debería importar nada de ellos. Para compartir código entre componentes hay que mover cualquier función o clase compartida a un archivo .ts diferente.

```
// src/utils/utils.ts
export function format(first: string, middle: string, last: string): string {
  return (first || '') + (middle ? ` ${middle}` : '') + (last ? ` ${last}` : '');
}

// src/components/my-component/my-component.tsx
import { Component, Prop, h } from '@stencil/core';
import { format } from '../../utils/utils';
@Component({
  tag: 'my-component',
```

Tipado de Componentes

- Los Web Components generados con Stencil vienen con archivos de declaración de tipo generados automáticamente por el compilador de Stencil.
- En general, las declaraciones de TypeScript brindan solidas garantías al consumir componentes:
  - Garantizar que los valores adecuados se transmitan como propiedades.
  - Autocompletado de código en IDE modernos como VSCode
  - Detalles de eventos
  - Firmas de los métodos de los componentes
- Estos tipos públicos son generados automáticamente por Stencil en formato src/component.d.ts. Este archivo permite una escritura solida del JSX (al igual que React) e interfaces HTMLElement para cada componente. Se recomienda que este archivo se registre con el resto de su código en el control de código fuente.
- Debido a que los componentes web generados por Stencil son vanilla Web Components, amplían la interfaz HTMLElement. Para cada componente se registra en el ámbito global un tipo nombrado HTML{PascalCaseTag}Element. Esto significa que los desarrolladores NO tienen que importarlos explícitamente, al igual que HTMLElement o HTMLScriptElement no se importan.
- El JSDoc del código con @Component, @Prop, @Method y @Event permite la generación de documentación y una mejor experiencia de usuario en un editor que tiene inteligencia de TypeScript.

Muro de imágenes