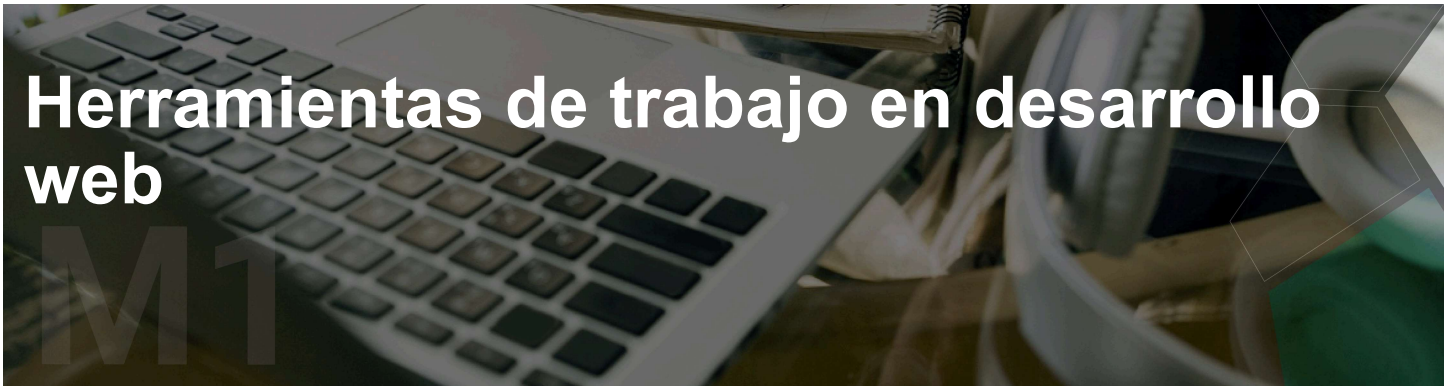


Herramientas de trabajo en desarrollo web

M1

Introducción

Utilizar un software de control de versiones es prácticamente una obligación para cualquier desarrollo que vaya a llevar más de quince minutos de trabajo. No solo va a permitirnos gestionar las diferentes versiones del código, sino que también nos va a hacer posible el trabajo en equipo sobre las mismas piezas de software. En esta lectura, presentaremos Atom, un entorno de desarrollo integrado (IDE, por las siglas en inglés, integrated development environment), que es un tipo de software que nos facilitará siempre el trabajo a la hora de programar.

1. GIT

Supongamos que formamos parte de un equipo de trabajo y estamos desarrollando el software para La Siglo. Empezamos a desarrollar las bases del panel de administración, avanzamos unos días en el desarrollo.

Situación 1. De pronto, probamos una librería adicional para resolver la autenticación y nos arroja un error. Intentamos revertirlo, pero *es muy difícil recordar los diferentes archivos que modificamos.*

Situación 2. Dos personas diferentes del equipo han trabajado en diferentes funcionalidades, haciendo muchas modificaciones en diversos archivos. Cuando ambas terminan, deben juntar todo en un único proyecto. La verdad es que *no recuerdan muy bien qué líneas de código han modificado ni todos los archivos usados.*

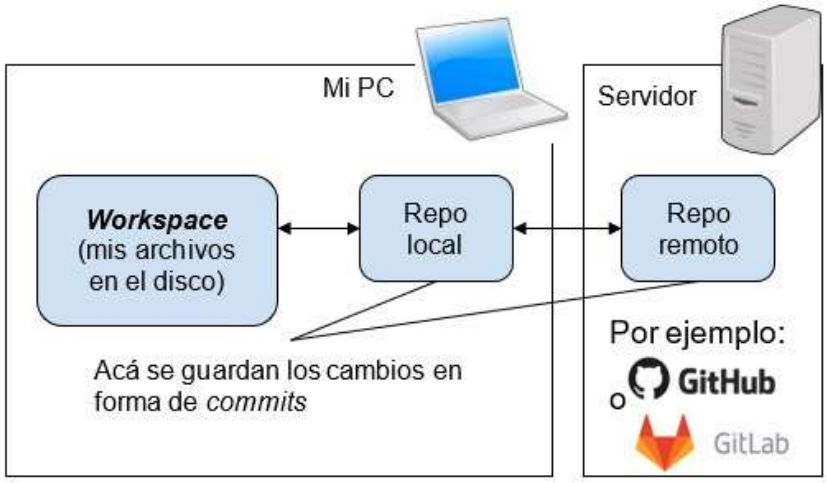
Para resolver estas dos situaciones, y un sinfín de otras posibles en el trabajo en equipo al desarrollar software, **recomendamos enfáticamente el uso de un software de control de versiones (VCS, por las siglas en inglés de version control software) y proponemos GIT en particular.** Nos adentraremos en esta herramienta siguiendo la presentación que realizan Sandler y Sanzo (2017), en un apartado de la documentación de Wollock:

Git es un software de control de versiones que facilita mucho el trabajo de programar en un equipo de personas. En particular, resuelve varios problemas: los que tienen que ver con compartir el código para que varios desarrollen, y permitir tener control sobre los cambios. ¡Es como tener un Ctrl+Z con esteroides!

Entonces, lo que hace GIT es **agregar dos lugares más donde va a estar tu código, además, del típico lugar en disco:** esos lugares se llaman ***repositorios***. GIT básicamente trabaja con dos repositorios:

- **Repositorio local:** almacena las modificaciones que se fueron haciendo dentro de la máquina.
- **Repositorio remoto:** almacena una versión estable del sistema que se está desarrollando. ¡Es donde va a estar el código accesible para todos! (p. 1)

Figura 1. Estructura del sistema de versionado GIT



Fuente: Sandler y Sanzo, 2017, p. 1

Sandler y Sanzo (2017) continúan explicando que:

Una vez que los cambios realizados de forma local están funcionando correctamente, entonces, se los sube al servidor remoto para que puedan ser compartidos, permitiendo que otras personas dentro del mismo grupo de desarrollo puedan acceder a esas modificaciones.

Cada usuario va trabajando dentro de su repositorio local (modificando, agregando o eliminando archivos), y una vez que se llega a un punto estable (no rompí nada de lo que había y lo que agregué funciona bien), entonces, es un buen momento para subirlo al repositorio remoto. Para esto, hay que seguir los siguientes pasos:

- **Elegir qué cambios de los que se hicieron en los archivos se quieren guardar en el repositorio local.**
- **Hacer un commit:** esto guarda los cambios en el repositorio local.
- **Hacer un push:** esto pasa lo que hay guardado en el repositorio local al repositorio remoto. Este es el único paso que requiere conexión a internet, ya que envía información, mientras que los dos primeros pasos son locales. (p. 1)

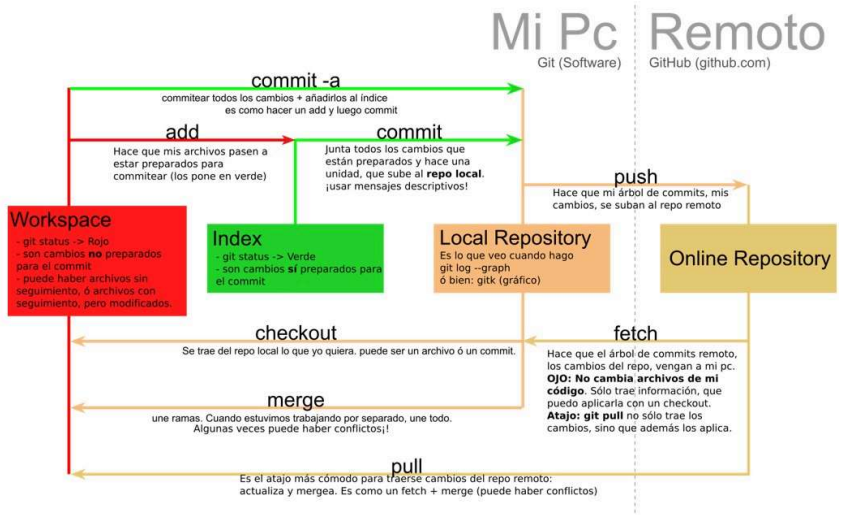
Cabe mencionar que para usar GIT existen varios software, *tanto de interfaz gráfica como por línea de comando*; ya que muchos implementan la función de ser clientes de esta herramienta. **Aquí nos enfocaremos en dos software:**

- **Por línea de comando,** por ser siempre la más directa y transparente, independiente del stack tecnológico y herramientas utilizadas.
- **A través del IDE,** para que puedas aprovechar las ventajas del VCS sin salirte del editor que estás utilizando.

Te recomiendo que crees una cuenta en **GitHub** para empezar a trabajar de esta manera. Luego, descarga el cliente por terminal para tu sistema operativo: https://git-scm.com/downloads.

Sandler y **Sanzo** (2017) presentan una síntesis de los comandos más comunes que utilizaremos. Cabe destacar que la terminología, **al utilizar un cliente de GIT integrado al IDE**, es idéntica a los comandos, de modo que resulta interesante conocerla indistintamente de la forma de utilizarla.

Figura 2. Terminología completa de GIT



Fuente: Sandler y Sanzo, 2017, p. 29

Lo primero que vamos a hacer es ***configurar el usuario de GIT*** en nuestra computadora a través de la terminal:

```
git config --global user.email "micuenta@gmail.com"
git config --global user.name "Nombre Apellido"
```

Luego, para evitar autenticarte cada vez que intentes acceder al repositorio remoto, **se recomienda la autenticación por SSH.** Para conocer cómo hacerlo, contamos con *documentación oficial* de GitHub: https://docs.github.com/es/github/authenticating-to-github/connecting-to-github-with-ssh.

Hay diferentes formas de trabajar con GIT en un proyecto. **Una es crearlo en la plataforma primero (en GitHub); luego clonarlo localmente (para crear el repositorio local) del siguiente modo:**

```
git clone [url]
```

En esa carpeta, empezar a trabajar en el proyecto. ***Esta sirve si aún no existe nada de código del proyecto.***

Otra forma es crear el repositorio remoto es inicializarlo localmente:

```
git init
```

Luego, debemos indicarle localmente cuál es su remoto (la URL del repositorio que nos brinda GitHub, que finaliza en .git):

```
git remote add origin [url]
```

Una vez que disponemos del proyecto localmente, cada vez que estamos en un punto estable, agregamos los cambios:

```
git add [rutas de archivos o . para todos los cambios]
```

Y luego los *commiteamos* al repositorio local:

```
git commit -m “Un mensaje que describa qué cambios hicimos”
```

En el mensaje de *commit*, **es muy importante que no comentes cuáles archivos o líneas de código modificaste, esa información va a estar disponible, al igual que fecha, hora y autor del commit**. El texto *debe comunicar de forma declarativa qué se hizo y no cómo se hizo*. El consejo es que no uses mensajes muy generales, al estilo «**solucioné errores**», porque a futuro no va a servirte esa información. Ya sea un ***bugfix***, la corrección de un typo o una funcionalidad nueva, explicítala lo mejor posible.

Una vez que tenemos el commit hecho al repositorio local, debemos pushearlo:

```
git push origin master
```

Si es la primera vez que pusheamos a un repositorio que acabamos de crear y vincular con GIT remote add, debemos pushear de esta manera:

```
git push -u origin master
```

¿Qué es eso de master? GIT nos permite el trabajo por ramas o branches. *Si es un proyecto o equipo mediano, es una práctica muy recomendable.* Cada **branch**, como las ramas de un árbol, se abre desde la principal, que es master, para trabajar en algo puntual. **Por ejemplo**, una funcionalidad nueva. Se realizan varios commits en ese camino al repositorio remoto, y cuando se termina la funcionalidad, se mergea esa branch a master nuevamente, aplicando todos los cambios realizados. **En el medio, en master se siguió trabajando normalmente**.

Muchos equipos de trabajo tienen en master la versión actual de producción, la que está desplegada en el entorno real, y utilizan las branches para avanzar en desarrollos nuevos, bugfixes, etcétera.

Pueden ocurrir conflictos si hay dos versiones que mergear que han modificado en la misma parte de los mismos archivos, tanto de diferentes branches uniéndose como en una misma por diferentes desarrolladores. Por suerte, contamos con **herramientas gráficas que nos ayudarán a ver y resolver estos conflictos de forma sencilla**.

Hay un sinfín de flujos de trabajo posibles con GIT y *aquí solo hemos abordado el más usual*, pero contamos con que la comunidad de GIT, que es realmente **amplia** y **colaborativa**, nos permitirá encontrar ayuda cuando queramos tomar caminos más elaborados o resolver conflictos. **La recomendación es ir de a poco: comenzar por una única branch en algún proyecto**

unipersonal, para conocer el flujo más sencillo, y luego lanzarte a resolver los más complejos, en la medida en que los necesites.

2. IDE

Elegimos trabajar con un entorno de desarrollo integrado (IDE) en lugar de un mero editor de texto, ya que, si bien no precisamos en el caso de PHP de tareas de compilación, el autocompletado inteligente de código, resaltado de sintaxis y un sinfín de plugins open source disponibles en la web, nos harán el trabajo muchísimo más fácil.

Atom es un IDE que resulta muy **flexible** y **configurable**, y como fue desarrollado por GitHub, *tiene una integración muy confortable con esta plataforma*.

Integración IDE con GIT

Para el uso cotidiano, es muy práctico utilizar un único *software* que se integre con los demás de forma transparente. Por eso, **te recomendamos que uses GIT a través de Atom.**

Al igual que por terminal, **hay distintas formas de lograr nuestros objetivos**, dependiendo del flujo de trabajo que tengamos o elijamos. Abordaremos los pasos para uno posible, con un proyecto nuevo creado que aún no ha sido subido al remoto, y contamos con la amplia documentación en línea de estas herramientas si quisiéramos seguir otro flujo.

Lo primero es crear el repositorio local en Atom, teniendo abierta la carpeta de nuestro proyecto.

Figura 3. Creación de un repositorio local en Atom

Fuente: captura de pantalla de Atom (GitHub, 2017)

Luego, seleccionamos la ubicación del repositorio (por defecto, la carpeta abierta):

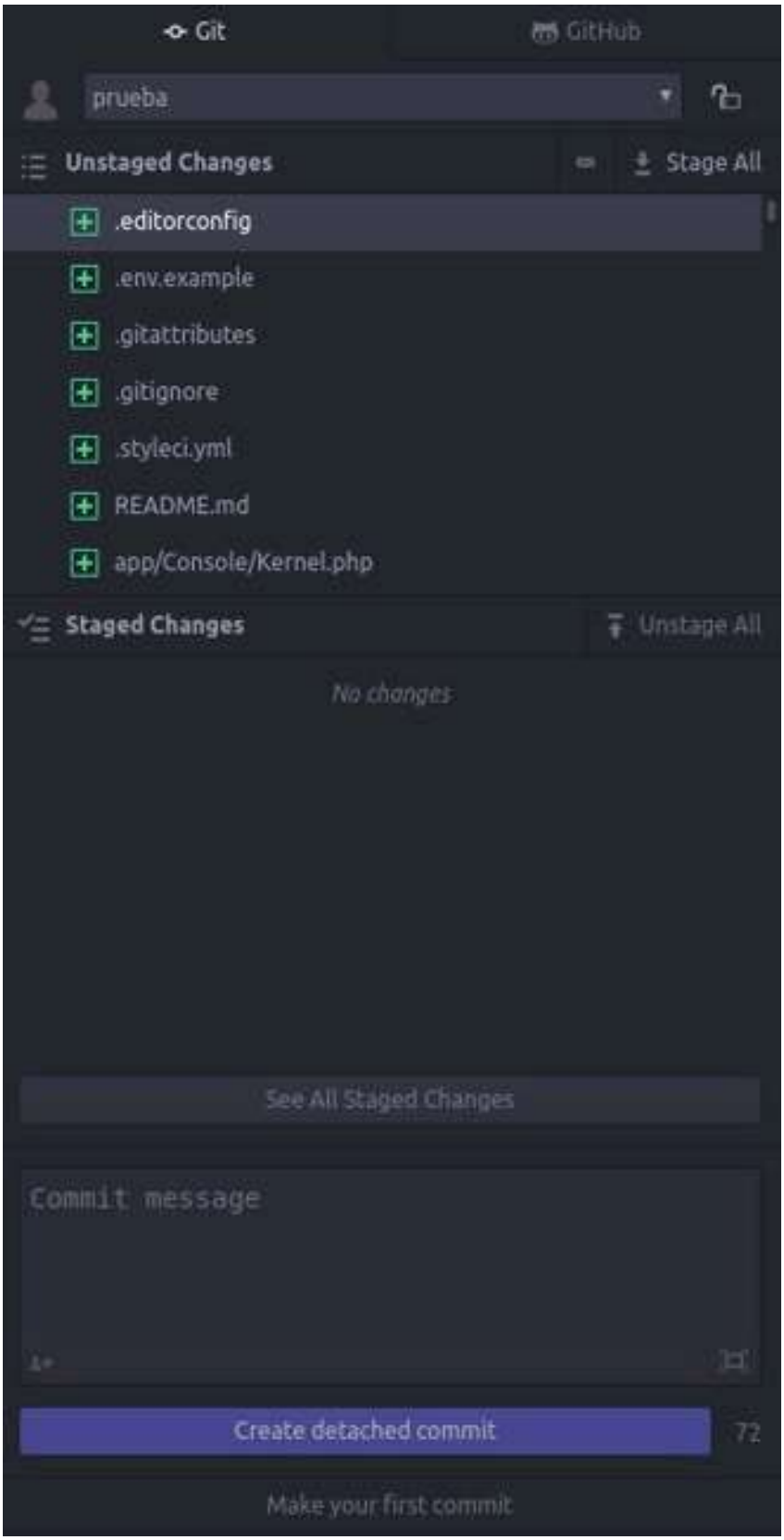
Figura 4. Selección de la ubicación del repositorio local en Atom

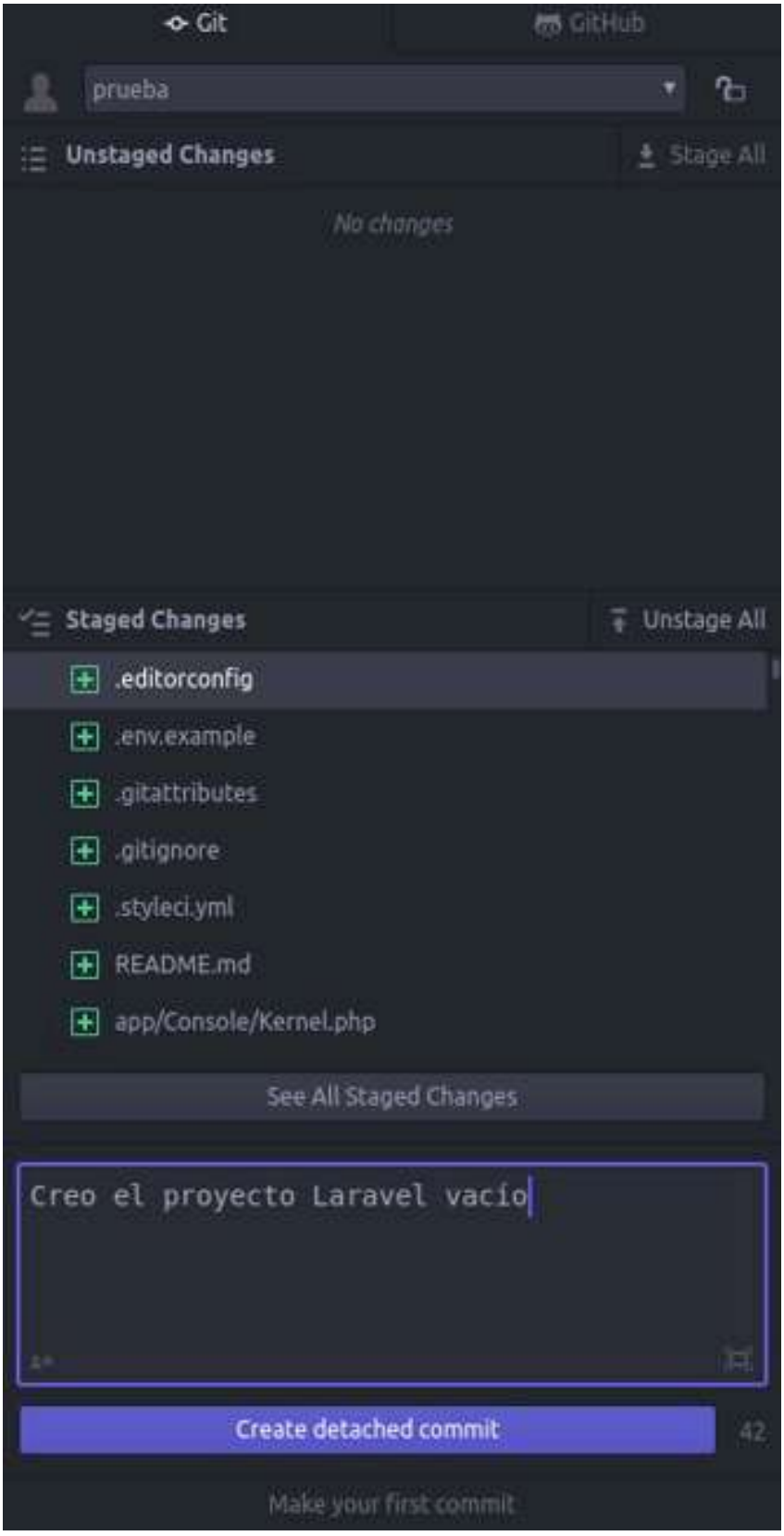
Fuente: captura de pantalla de Atom (GitHub, 2017)

Esto ejecutará el comando «**git init**». Luego, podemos hacer nuestro primer **commit detached**, separado de un repositorio remoto que aún no configuramos. Para eso, **agregamos al stage los**

archivos que queremos incluir (en la imagen se hizo un stage all) **y colocamos un mensaje de commit:**

Figura 5. Crear commit detached





Fuente: captura de pantalla de Atom (GitHub, 2017)

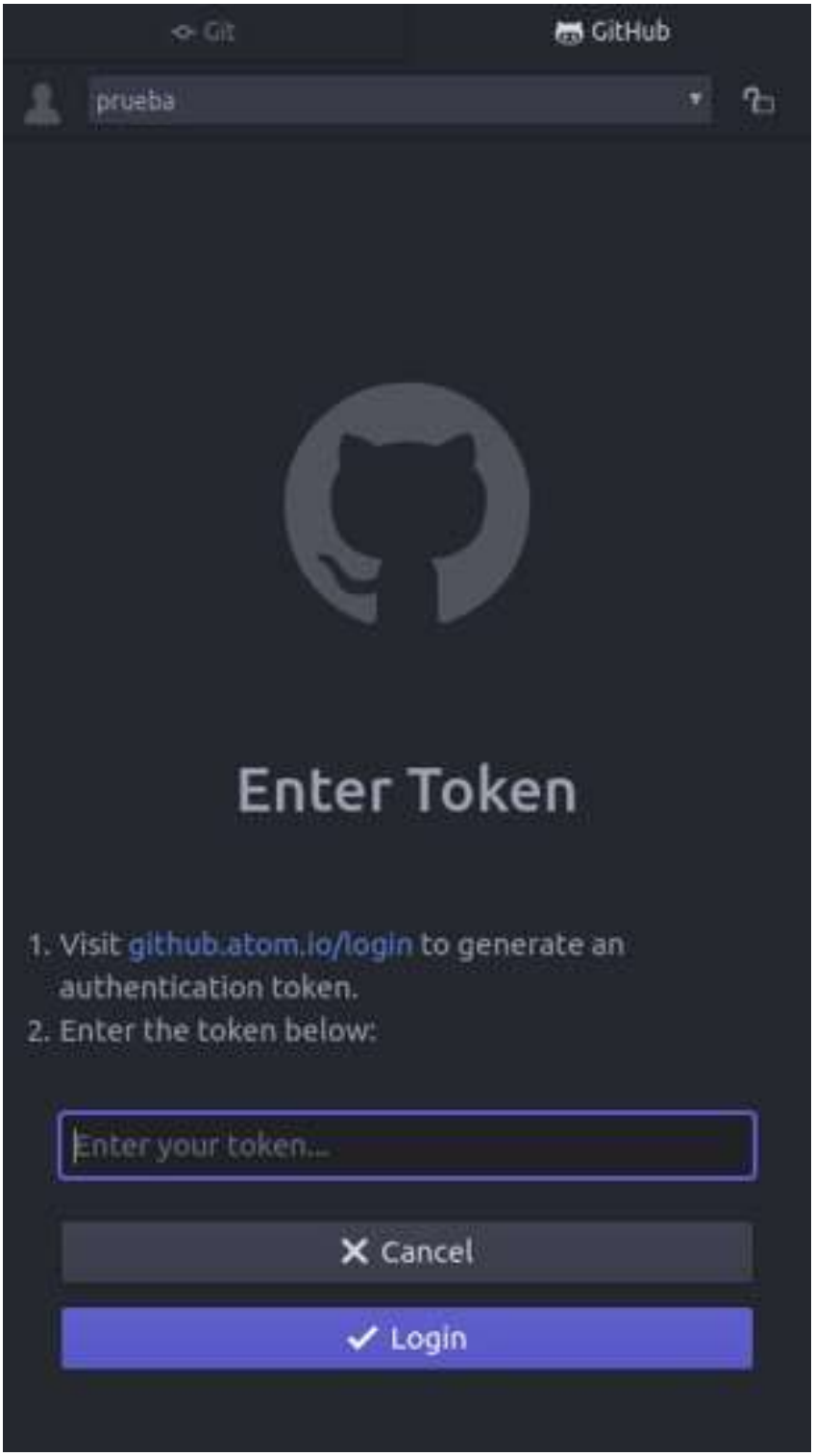
Luego, vamos a modificar el archivo de configuración de GIT, en «**.git/config**», agregando la URL del repositorio remoto (**se hizo utilizando SSH, puede hacerse con https también**) y la branch master:

Figura 6. Agregar URL del repositorio

Fuente: captura de pantalla de Atom (GitHub, 2017)

Una vez realizado el paso anterior, **guardamos y vamos a la solapa de GitHub en Atom**, donde se nos va a pedir que generemos un **token**, autorizando la aplicación en el *browser* y que copiemos ese *token*.

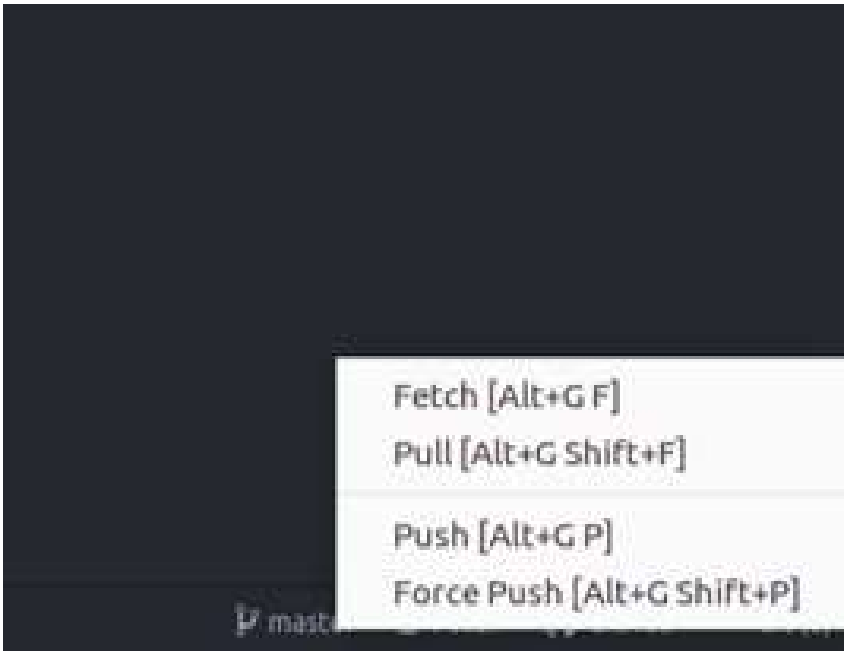
Figura 7. Autorizar la aplicación en el browser con un token



Fuente: captura de pantalla de Atom (GitHub, 2017)

Por último, podemos hacer clic derecho en la opción «**Fetch**», que aparece abajo en la izquierda de la ventana del Atom, y hacer el push.

Figura 8. Hacer el push



Fuente: captura de pantalla de Atom (GitHub, 2017)

En sucesivos **commits**, **push** y **pull**, una vez configurado lo necesario, *podremos solo manejarnos con la solapa de GIT y estas opciones.*

Referencias

GitHub, (2017). *Atom* [entorno de escritorio]. GitHub

Sandler, T. y Sanzo, A. (2017). *Tutorial Git en Wollock.* Versión 2.0. https://docs.google.com/document/d/1p4W1wwzzdvzfdGbvXexbE3arwyAAg1xirYW68Twkatc/edit