2. Git

2.1. Introducción

En este apartado presentaremos Git, uno de los sistemas de control de versiones de código más populares. En primer lugar se introducirá el concepto de sistema de control de versiones y, a continuación, se presentarán los conceptos y comandos básicos de Git.

Nota

En este apartado, revisaremos los conceptos y comandos de Git que os resultarán más útiles durante el desarrollo de la asignatura. Sin embargo, por cuestiones de limitación de tiempo y espacio, no podemos ver en profundidad todos los conceptos y características de Git. En cualquier caso, siempre podéis consultar la documentación oficial de Git en el siguiente enlace: https://gitscm.com/.

2. Git

2.2. Sistemas de control de versiones

El desarrollo de sistemas software es un proceso evolutivo e incremental donde uno o más desarrolladores crean el código del producto. Su carácter incremental y la participación de varios actores para la creación del sistema motivan la necesidad de disponer de sistemas de control de versiones para gestionar los cambios que se producen en el código. Estos sistemas han de facilitar la administración de las distintas versiones del producto, así como de cada uno de sus componentes.

En general, un sistema de control de versiones ha de proporcionar lo siguiente:

- **Solución de almacenamiento** para los artefactos software del sistema en desarrollo, es decir, los diferentes ficheros que conforman nuestro sistema.
- **Registro histórico** de las versiones de cada artefacto software.
- Registro de los **cambios individuales** (añadir, eliminar, reemplazar) entre cada par de versiones.

En un sistema de control de versiones, llamamos **repositorio** al lugar donde se almacena el conjunto de ficheros del sistema software que estamos desarrollando, así como la información histórica de sus componentes. Cada una de las revisiones de nuestro sistema se llama **versión**, y algunas de ellas pueden etiquetarse utilizando una ***tag***, por ejemplo, una determinada revisión puede etiquetarse con la *tag* `v1.0` cuando alcanzamos la primera versión estable de nuestro sistema.

> **Nota**
>
> En los últimos años se ha popularizado el uso de etiquetas para denominar versiones de un producto software que siguen el formato llamado **versionado semántico** (o *semantic versioning*, en inglés). Este formato propone el uso de etiquetas para denominar las versiones de un producto software siguiendo el formato X.Y.Z, donde los valores que pueden tomar X, Y y Z son numéricos y representan la versión mayor, menor y parche, respectivamente. Se puede encontrar más información sobre el versionado semántico en la página oficial: https://semver.org/lang/es/.

La propia evolución de un repositorio y sus revisiones genera una estructura lineal, donde cada revisión tiene una revisión anterior y posterior (excepto la última revisión, que no tiene revisión posterior). Durante el desarrollo, es posible que una revisión se bifurque en dos revisiones diferentes, creando una **rama** (*branch*, en inglés). Cada rama puede evolucionar de forma independiente (con nuevas revisiones) y unirse a otras ramas en un futuro (mediante el procedimiento de *merge*, en inglés). De esta forma, la evolución de un repositorio a lo largo de las revisiones y ramas suele tener una estructura de árbol.

Los sistemas de control de versiones se pueden clasificar en los siguientes:

- **Distribuidos**. En estos sistemas, cada miembro del equipo de desarrollo tiene su propio repositorio que se sincroniza con el resto de los miembros del equipo. Normalmente se identifica uno de los repositorios como el punto central de sincronización. Git se basa en esta aproximación, pero también Mercurial (que no veremos en esta asignatura).
- **Centralizados**. En estos sistemas, existe un repositorio centralizado al cual se conectan los miembros del equipo de desarrollo para obtener el código, enviar sus cambios y, en general, realizar todas las tareas relacionadas con la gestión del código. Algunos ejemplos son CVS o Subversion (que no veremos en esta asignatura).

2. Git

2.3. Conceptos básicos de Git

Git es un sistema de control de versiones diseñado por Linus Torvalds y lanzado en el año 2007.

La figura 2 muestra una captura de una parte de la página web oficial de Git.

Su aparición surgió como respuesta a los sistemas populares en aquellos años, que eran fundamentalmente CVS y Subversion, con el objetivo de ofrecer un sistema de control de versiones eficiente y distribuido.

Nota

Durante este subapartado complementaremos las explicaciones con enlaces a la documentación oficial de Git, pero también te recomendamos revisar otras fuentes, como por ejemplo el libro Pro Git, cuyo contenido está disponible en línea con licencia Creative Commons.

Figura 2. Página oficial de Git
(capturada en septiembre de 2022).

Construido como un sistema de control de versiones distribuido, cada miembro del equipo de desarrollo mantiene un repositorio local, el cual se puede sincronizar con los repositorios del resto de miembros. En la práctica, es común ofrecer un repositorio remoto central, donde todos los miembros del equipo sincronizan sus versiones. En la actualidad, estos repositorios remotos centrales suelen ser accesibles en línea, por ejemplo, utilizando plataformas como GitHub.

Un repositorio Git almacena los cambios realizados en un conjunto de ficheros del sistema y los registra acorde a diferentes versiones. De esta manera, un repositorio Git parte de una versión inicial, que contiene el conjunto de ficheros iniciales cuyas versiones se quieren gestionar, y registra revisiones de dichos ficheros. Estas nuevas revisiones de los ficheros se denominan **versiones** y cada una de ellas se crea mediante un **commit**, identificado por un valor único, comúnmente llamado `sha` . De esta manera, se puede obtener una determinada versión de un repositorio recuperando el commit correspondiente.

La figura 3 muestra un repositorio con diferentes commits. Cada commit está representado con un círculo. El commit inicial en la figura tiene el *sha* `c1` , y el resto de commits (de `c2` a `c6`) contienen la evolución de los ficheros del repositorio. De forma gráfica, se suelen unir al conjunto de commits relacionados mediante una línea para facilitar la comprensión de la evolución temporal en el repositorio.

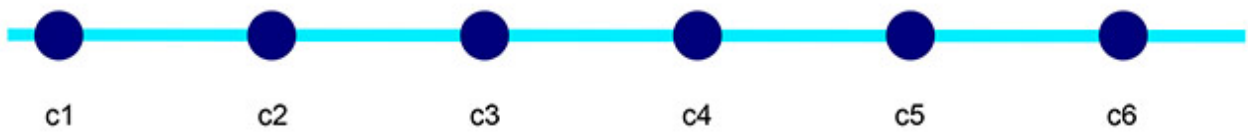

Figura 3. Representación de un repositorio Git con 6 commits
(capturada en septiembre de 2022).

Un commit contiene los cambios hechos a un conjunto de ficheros del repositorio. De esta manera, es posible que un commit no contenga cambios para un determinado fichero, en este caso, el estado del fichero no ha cambiado y se puede recuperar consultando el commit anterior. Por ejemplo, en la figura 4, el commit inicial `c1` registra los cambios para los ficheros *README.md* y *script.js*. A continuación, el commit `c2` registra cambios solamente para el fichero *script.js*. Si se recupera el repositorio en el commit `c2`, el sistema de control de versiones recuperaría el fichero *README.md* a partir de la información del commit `c1` y el fichero *script.js* a partir de la información del commit `c2`.

Figura 4. Un repositorio con dos commits, `c1` y `c2`, donde el segundo commit solamente registra los cambios de uno de los ficheros.
(capturada en septiembre de 2022).

El conjunto de commits relacionados temporalmente (y representados gráficamente en una línea temporal) recibe el nombre de **rama**. A la rama principal del repositorio se le llama ***main*** (antiguamente llamada **master**). Un repositorio puede contener cualquier número de ramas. El hecho de trabajar en ramas diferentes a la rama *main* permite al equipo de desarrollo realizar cambios en los ficheros de forma individual.

Por ejemplo, en un proyecto de desarrollo de una página web, el equipo de diseñadores puede trabajar en una rama distinta a la que utiliza el equipo que se encarga de implementar la funcionalidad lógica de la web. Otro ejemplo sería crear ramas separadas para el *front-end* y *back-end* de una aplicación.

Dado que las ramas evolucionan de manera independiente, es decir, contienen sus propios commits para registrar los cambios en los ficheros, suele ser común alinear las versiones con otras ramas para sincronizar los cambios. A la acción de sincronizar dos ramas se denomina ***merge***. A continuación veremos varios ejemplos para visualizar el concepto de rama y la acción de *merge*.

En la figura 5a se pueden observar dos ramas, *main* y *dev*, siendo la segunda la rama utilizada por el equipo de desarrollo para implementar y probar nuevas funcionalidades del sistema. Con este esquema se asegura que la versión registrada en la rama *main* es la estable. Obsérvese la acción de *merge* de la rama *dev* a la rama *main* para incorporar los cambios desarrollados en la rama principal.

La figura 5b ilustra un escenario donde la rama *main* registra los cambios estables de una aplicación web en desarrollo, mientras que las ramas *front-end* y *back-end* implementan los correspondientes componentes de dicha aplicación. Como puede observarse, hay una acción de *merge* de la rama de *front-end* a *main*, la cual incorporaría los cambios de dicho componente; mientras que hay dos acciones de *merge* de la rama *back-end* a *main*, que ilustran un posible escenario donde se han realizado cambios de última hora. En este escenario los cambios afectan a ficheros diferentes, el siguiente ejemplo ilustra un escenario donde los cambios pueden afectar a los mismos ficheros.

La figura 5c muestra un escenario donde dos miembros del equipo desarrollo (llamados *developer 1* y *developer 2*) hacen evolucionar los ficheros. Para mantener su trabajo actualizado, aplican acciones de *merge* durante el desarrollo y, finalmente, a la rama principal.

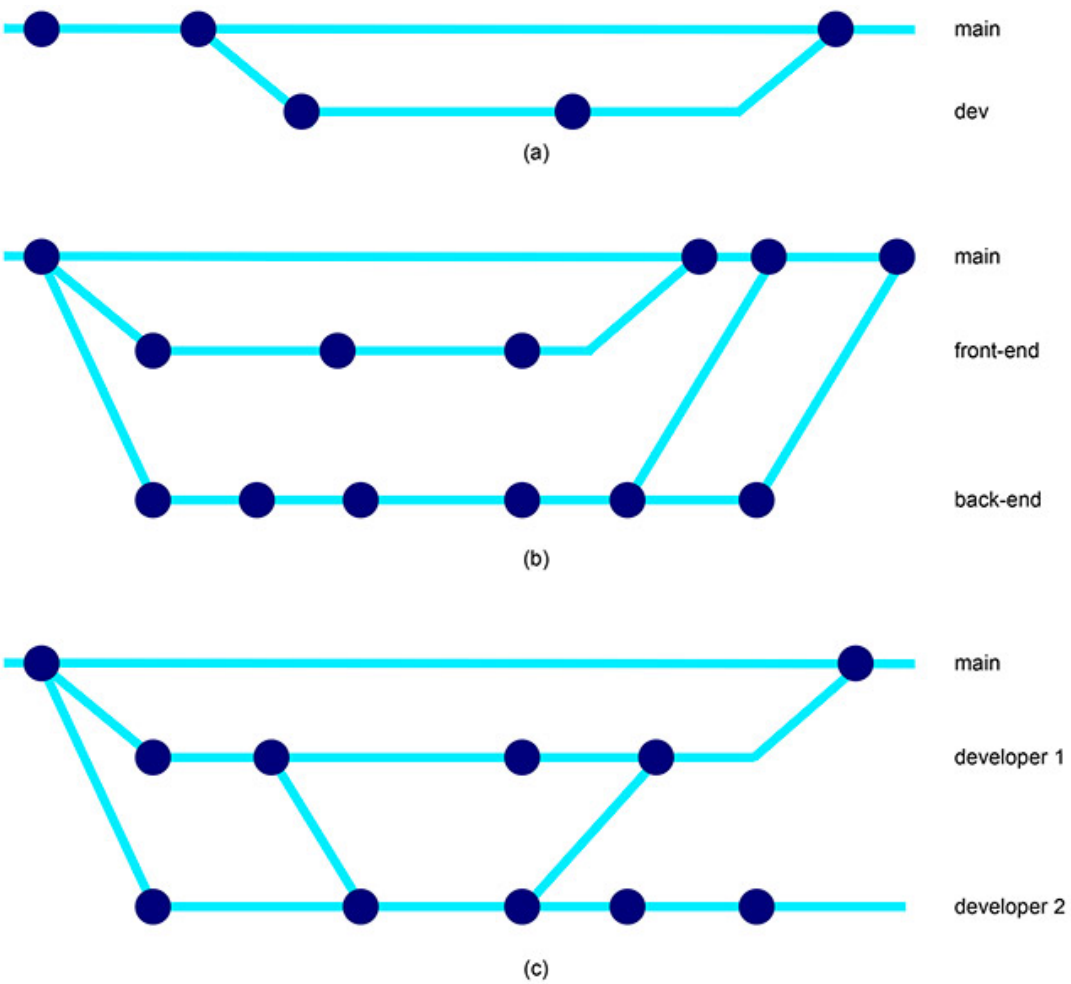

Figura 5. Ejemplos de ramas y acciones de *merge*

Hasta ahora se han ilustrado los conceptos suponiendo un repositorio centralizado donde conviven los commits y ramas. Sin embargo, Git es un sistema distribuido donde cada miembro dispone de su propio repositorio. De esta forma, los commits y ramas se crean inicialmente en el repositorio local, y es el miembro del equipo de desarrollo el que debe sincronizar determinadas ramas con otros repositorios remotos.

La acción de enviar una rama desde un repositorio local a uno remoto se denomina `push`, mientras que la acción contraria, es decir, recuperar una rama desde un repositorio remoto a uno local, se denomina `pull`.

Un despliegue típico de repositorios Git consiste en un repositorio central que almacena la rama *main*, así como otras que se utilicen para colaborar entre miembros del equipo de desarrollo (en los ejemplos de la figura 5, las ramas *dev*, *front-end*, *back-end*, *developer 1* y *developer 2*). Por ejemplo, en el caso del ejemplo de la figura 5a, cada miembro del equipo de desarrollo realizaría las siguientes acciones:

- un pull para recuperar los últimos cambios del repositorio central,
- commits para registrar sus cambios,
- un push para enviar los cambios al repositorio y así permitir que otros desarrolladores actualicen sus repositorios.

Nota

En Git es posible indicar qué ficheros no deben ser monitorizados, es decir, qué ficheros se ignorarán. Para ello, se debe crear un fichero llamado *.gitignore* que incluya el nombre de los ficheros que deben ignorarse.

Cada línea del fichero debe indicar el nombre del fichero indicando su ruta a partir de la raíz del repositorio. Por ejemplo, un fichero llamado *README.md* y localizado en la raíz del repositorio se indicará como *README.md*; mientras que un fichero llamado *test.js* localizado en la carpeta *js* se indicará como *js/test.js*. También se pueden utilizar comodines, por ejemplo, para ignorar todos los ficheros con la extensión *.temp* se puede indicar **.temp*.

La documentación oficial da más información sobre el formato de este fichero y puede consultarse en: https://git-scm.com/docs/gitignore.

2. Git

2.4. Comandos básicos de Git

En este subapartado haremos un repaso de los comandos de Git más importantes y que generalmente se utilizan en el día a día de un equipo de desarrollo. Para cada comando se describe su funcionamiento y se incluyen algunos ejemplos.

> **Enlace recomendado**
>
> Puede consultarse la documentación oficial en el siguiente enlace: https://gitscm.com/docs/.

1) El comando `init`. El comando `init` inicializa un repositorio en el sistema de ficheros. Es el primer caso que se realiza cuando se crea un repositorio en una máquina local. Si en un futuro se desea crear un repositorio central (para poder realizar las acciones de `push` y `pull`), se ha de configurar el repositorio mediante el comando `config`). Toda la información necesaria para gestionar el repositorio Git se encuentra en el directorio *.git*.

> **Enlace recomendado**
>
> Para la creación de un repositorio central mediante el comando `config`, que no veremos en estos materiales, puede consultarse la documentación oficial en el siguiente enlace: https://git-scm.com/docs/git-config.

Ejemplo:

```
git init
```

> **Enlace recomendado**
>
> La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/ git-init.

2) El comando `clone`. El comando `clone` realiza una copia de un repositorio Git remoto en nuestro sistema de Este es generalmente el paso inicial para poder trabajar con los ficheros de un repositorio Git. Además de copiar los ficheros remotos en nuestro sistema, también inicializa la información básica necesaria para gestionar el repositorio Git (que se encuentra en el directorio *.git*).

Ejemplo:

```
git clone https://github.com/PJP-UOC/repository
```

> **Enlace recomendado**
>
> La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-clone.

3) El comando `add`. El comando `add` añade un nuevo fichero, o uno existente pero que ha sido modificado, para registrar sus cambios en el repositorio Git. Este comando se ha de ejecutar siempre que se modifiquen ficheros del repositorio y se desee

registrar sus cambios por el sistema de control de versiones.

Es importante tener en cuenta que este comando actualiza nuestro repositorio local para que considere el o los ficheros en la próxima versión. Para hacer efectivo el registro de cambios en el o los ficheros se ha de ejecutar el comando `commit`, que veremos a continuación.

Ejemplos:

```
git add README.md
git add PEC1.js
git add js/script.js
```

> **Enlace recomendado**
>
> La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-add.

4) El comando `commit`. El comando `commit` incorpora los ficheros añadidos en el repositorio (véase comando `add`) al repositorio del sistema. Este comando hace evolucionar el repositorio a una nueva versión, ya que una vez se han *comiteado* los ficheros, los cambios se registran en el sistema de control de versiones como una nueva versión. Para cada `commit` se debe indicar un comentario breve de los cambios introducidos.

> **Nota**
>
> Es muy recomendable acostumbrarse a escribir los comentarios de los commit en inglés, ya que es el lenguaje comúnmente utilizado en los equipos de desarrollo.

Ejemplos:

```
git commit -m "Changes in the README.md"
git commit -m "Added PEC1.js"
git commit -m "Updated script.js with last exercise"
```

> **Enlace recomendado**
>
> La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-commit.

5) El comando `push`. El comando `push` envía los cambios registrados por las diferentes versiones del repositorio local (véase comando `commit`) al repositorio remoto. Si se trabaja con GitHub, este repositorio remoto estará alojado en dicha plataforma.

Es importante recordar que se enviará la última versión *comiteada* del repositorio local, por lo que es importante verificar que si se han realizado cambios en los ficheros se ha de haber hecho un `add` y `commit` anteriormente de dichos ficheros.

Ejemplo:

```
git push origin main
```

Enlace recomendado

La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-push.

6) El comando `pull` . El comando `pull` recupera todos los cambios del repositorio remoto (en el caso de utilizar GitHub, el repositorio remoto estará alojado en esta plataforma) y los incorpora al repositorio local del sistema. Este comando puede verse como la tarea contraria al comando `push` y se utiliza para recuperar los cambios que hayan podido enviar otros miembros del equipo de desarrolladores al repositorio remoto.

Ejemplo:

```
git pull
```

Enlace recomendado

La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-pull.

7) El comando `status` . El comando `status` muestra el estado del repositorio en el Permite ver los cambios que se han preparado y los que no. Su funcionamiento se comprende de forma más clara ejecutándose antes y después del comando `add` .

Ejemplo:

```
git status
```

Enlace recomendado

La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-status.

8) El comando `log` . El comando `log` muestra el histórico de commits, es decir, el historial de todos los cambios que se han hecho en el código.

Ejemplo:

```
git log
```

Enlace recomendado

La documentación oficial de este comando se encuentra en: https://git-scm.com/docs/git-log.

2. Git

2.5. Ejemplo práctico de Git

En este subapartado presentamos un ejemplo práctico de creación de un repositorio Git y el uso de los comandos más relevantes.

En primer lugar, crearemos un repositorio llamado *myRepoTest*. En los siguientes listados de código, se invocarán los comandos Git en el directorio llamado *git* de un entorno Windows.

```
C:\git>git init myRepoTest
Initialized empty Git repository in C:/git/myRepoTest/.git/
```

El comando anterior ha creado el directorio *myRepoTest*, dentro del cual podremos crear los ficheros que queremos gestionar con Git. Como se puede observar, también se ha creado el directorio *myRepoTest/.git*, que contiene la información de gestión y configuración necesaria de Git para registrar los cambios.

A continuación, crearemos un fichero *README.md* en el repositorio, y procederemos a registrar sus cambios con `add` y `commit` .

```
C:\git\myRepoTest> git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
nothing added to commit but untracked files present(use "git add" to
track)
C:\git\myRepoTest> git add README.md
C:\git\myRepoTest> git commit - m "Adding the README.md"
[master(root - commit) 0 a784ee] Adding the README.md
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
C:\git\myRepoTest> git status
On branch master
nothing to commit, working tree clean
```

El fragmento de código anterior utiliza inicialmente el comando `status` para comprobar el estado del repositorio. Como puede observarse, Git indica que se está trabajando en la rama *master* y que no hay commits todavía. Además, lista los ficheros (*README.md*) cuyos cambios no se han registrado todavía en el repositorio.

A continuación, se ejecuta el comando add para indicar que se desea registrar los cambios en el fichero *README.md*.

Posteriormente, se utiliza el comando `commit` para incorporar los ficheros al repositorio. Como resultado de la ejecución de este comando, se muestra el identificador del commit (`0a784ee`) y los cambios introducidos.

Finalmente, se vuelve a ejecutar el comando `status` para comprobar que todos los cambios se han registrado en el repositorio.

Imaginemos que modificamos el fichero *README.md* y queremos registrar de nuevo sus cambios. El siguiente fragmento de código realiza esta acción:

```
C:\git\myRepoTest> git status On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified: README.md
no changes added to commit(use "git add" and / or "git commit -a")
```

```
C:\git\myRepoTest> git add README.md
C:\git\myRepoTest> git commit - m "Changes in README.md"
[master 304 b59f] Changes in README.md
 1 file changed, 1 insertion(+), 1 deletion(-)

C:\git\myRepoTest> git status
On branch master
nothing to commit, working tree clean
```

El código es muy parecido al fragmento anterior. La principal diferencia es que Git nos indica que ha habido modificaciones en el fichero *README.md*.

En cualquier momento podemos hacer uso del comando `log` para ver los últimos commits del repositorio:

```
C:\git\myRepoTest> git log
commit 304 b59f0c54ae7b00f5ed3bda0df57297c612529(HEAD - > master)
Author: Javier < me @jlcanovas.es >
Date: Thu Sep 29 09: 47: 56 2022 + 0200
    Changes in README.md
commit 0 a784eef6cc60f13820c65f493ab1f252983cf18
Author: Javier < me @jlcanovas.es >
Date: Thu Sep 29 09: 40: 55 2022 + 0200
    Adding the README.md
```

A modo de práctica, recomendamos trabajar con otras acciones de edición, creación y borrado de ficheros, para ejercitar el uso de los comandos de Git.

3. GitHub

3.1. Introducción

En este apartado presentaremos la plataforma GitHub, que se ha convertido quizás en la plataforma de desarrollo de proyectos de código abierto (*open source*) más relevante de los últimos años. En primer lugar veremos una breve introducción a este tipo de plataformas, a continuación describiremos el proceso de desarrollo propuesto en estas plataformas y finalmente los componentes principales de GitHub.

3. GitHub

3.2. Plataformas en línea de código abierto

La propia naturaleza del software de código abierto ha motivado, desde sus inicios, la necesidad de poner a disposición pública el código fuente. Además, también ha sido muy frecuente que el desarrollo de estos proyectos implique la colaboración de grupos de desarrollo dispersos geográficamente. Estas son dos razones principales que motivaron la creación de plataformas en línea donde tanto contribuidores como usuarios pudieran acceder al software de código abierto.

En los últimos años han aparecido diferentes plataformas que facilitan la publicación y desarrollo en línea de proyectos de código abierto. SourceForge, que se creó en 1999, es quizás una de las primeras plataformas que ofrecieron esta funcionalidad. En los años posteriores aparecieron otras alternativas, como GitHub, GitLab o BitBucket.

Este tipo de plataformas han popularizado (y casi estandarizado) el concepto de proyecto de código abierto y su presencia en internet. De forma general, un proyecto de código abierto se ofrece como un repositorio de código, e incluye un conjunto de herramientas de colaboración y comunicación para facilitar el desarrollo del proyecto. Precisamente por las herramientas de colaboración y comunicación, muchas veces a estas plataformas se les denomina *plataformas sociales de código*.

Entre las herramientas de colaboración más utilizadas se pueden encontrar las siguientes: un sistema de gestión de tickets (en inglés, *issue tracker*), foros de discusión, wikis o sistemas de gestión de *pull requests*. La última de las funcionalidades, referida a la gestión de *pull requests* es quizás la más característica, y por ello la veremos en mayor detalle en el siguiente subapartado.

3. GitHub

3.3. El desarrollo basado en *pull requests*

Las plataformas en línea de código abierto tienen como objetivo promover el desarrollo de software de código abierto. Para ello incorporan soluciones como los repositorios de código o las herramientas de colaboración y comunicación comentadas anteriormente. Una de las herramientas más particulares es la que permite la gestión de los denominados *pull requests*, que introducen un modelo de desarrollo propio de estas plataformas.

El modelo de desarrollo basado en *pull requests* (o *pull-based development model*, en inglés) es una propuesta de desarrollo software donde diferentes grupos de desarrolladores colaboran desde diferentes proyectos de código abierto a un proyecto principal.

Para su implementación, se utilizan los conceptos de *fork* y *pull request*, que veremos a continuación.

Un *fork* es una copia de un repositorio o proyecto de código abierto de un usuario (u organización) en otra cuenta de usuario (u organización). Por ejemplo, hacer un *fork* de un proyecto de otra cuenta a mi cuenta de usuario provoca que tenga una copia de dicho proyecto. La intención de esta acción es disponer de una copia del proyecto al que quiero contribuir sin necesidad de solicitar permisos de edición o de alterar el código del proyecto en la cuenta original.

Nota

La acción de poder realizar un *fork* de cualquier proyecto también da la posibilidad de utilizarlo para un fin diferente: hacer una copia de un proyecto y continuar su desarrollo de manera independiente al proyecto original.

A este tipo de *fork* se le denomina *hard fork* y suele darse cuando una comunidad está descontenta con el desarrollo de un proyecto de código abierto, lo que produce generalmente una escisión del equipo de desarrollo, que trabajará en el nuevo *fork* sin la intención de contribuir al proyecto inicial con un *pull request*.

Uno de los *hard forks* más conocidos fue el que se produjo en el proyecto *Node.js*, el cual produjo la creación de una nueva versión (o producto) denominado *io.js*.

Una vez hecho el *fork*, el equipo de desarrollo puede trabajar en el proyecto copiado implementando nuevas funcionalidades. Para llevar a cabo esta implementación, todas las modificaciones se realizan en una rama (o *branch*, en inglés) nueva, para así facilitar la identificación de los cambios. Una vez realizados los cambios, el equipo de desarrollo puede proponer incorporar sus modificaciones en el proyecto original mediante un *pull request*.

Un *pull request* es una acción mediante la cual la rama de un proyecto que es *fork* se envía al proyecto original para su evaluación como contribución externa. De esta manera, el equipo de desarrollo del proyecto original recibe una propuesta de cambios para el código fuente que está encapsulada en una rama (la rama donde trabajó el equipo de desarrollo que hizo el *fork*) y puede evaluar la propuesta sin necesidad de alterar el código del proyecto original. Un *pull request* suele ir acompañado de un hilo de discusión para evaluar la propuesta. Si finalmente el equipo de desarrollo del proyecto original decide incorporar los cambios del *pull request*, se hace un *merge* de la rama y se incorpora al proyecto.

La figura 6 ilustra el proceso de desarrollo basado en *pull requests*. En la figura 6a se muestra la acción de *fork*, con un repositorio original (*original repository*) con una rama llamada *main*, la cual contiene el commit `o1` ; y un repositorio resultado de haber hecho la acción de *fork* (*forked repository*), el cual contiene el commit original `o1` . En la figura 6b se muestra la acción de *pull request* donde el repositorio resultado del *fork* (*forked repository*) ha evolucionado (véanse los commits `f1` y `f2`) y se crea un *pull request* en el repositorio original con la rama *forked/main* (véase que la rama *forked/main* existe tanto en *forked repository* como en *original repository*). En este punto, el equipo de desarrollo del proyecto original puede revisar las contribuciones (commits `f1` y `f2`) y decidir si finalmente se incorporan al proyecto. Si la decisión es positiva, se produce el *merge* de la rama del *pull request* en la rama del proyecto, tal y como se muestra en la figura 6c (véase el *merge* de la rama *forked/main* sobre la rama *main* en *original repository*).

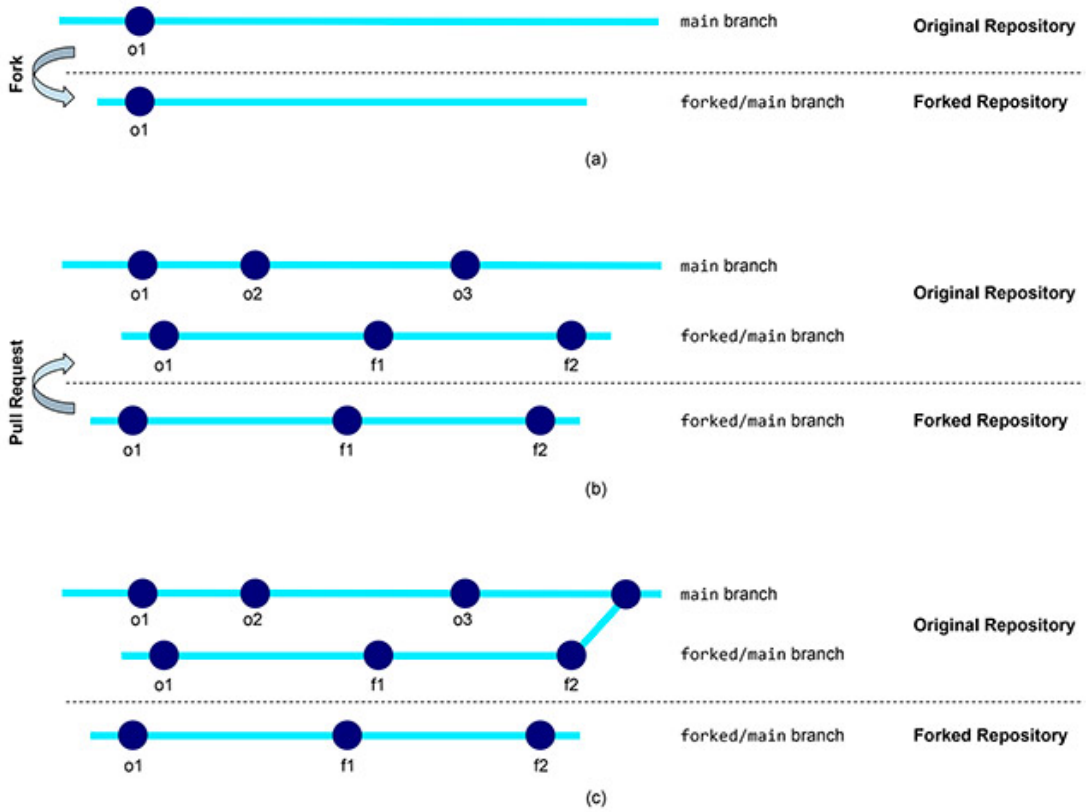

Figura 6. Ejemplo de proceso de desarrollo basado en *pull requests*

En general, el proceso de desarrollo basado en *pull requests* aprovecha los conceptos y funcionalidades de las plataformas en línea de código abierto para facilitar, de una manera eficiente y limpia, la colaboración entre los proyectos de la plataforma.

3. GitHub

3.4. Componentes de GitHub

GitHub es una plataforma de desarrollo colaborativo que se ha convertido en uno de los actores principales de la industria del desarrollo de software. En el año 2022, aloja a más de doscientos millones de repositorios y cuarenta millones de usuarios registrados, aunque estas cifras aumentan diariamente.

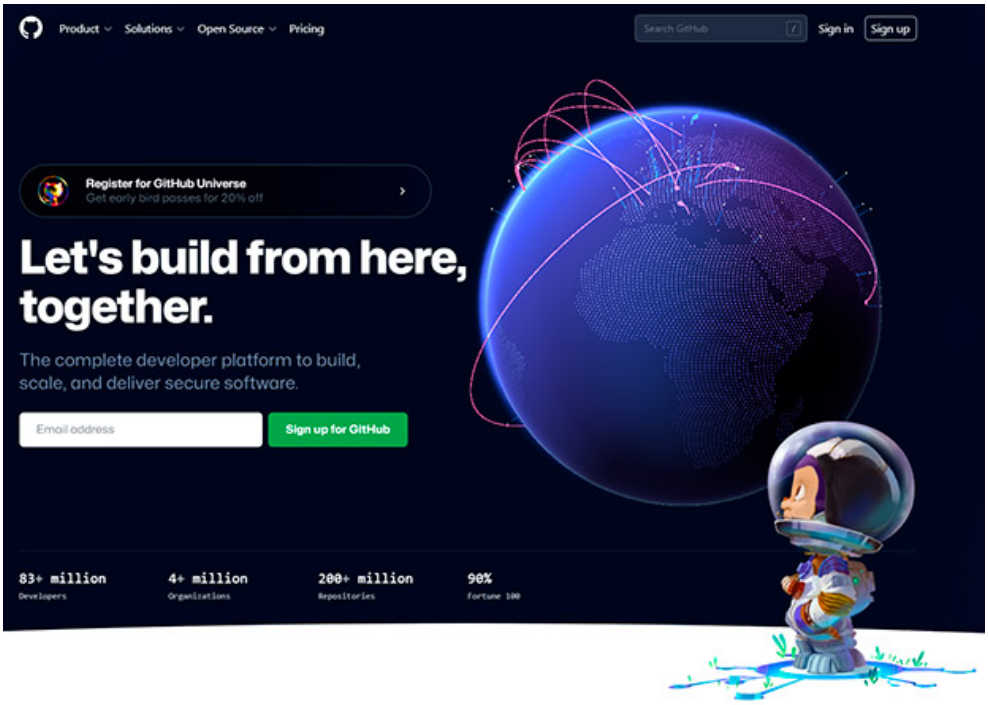

Figura 7. Página web de GitHub (capturada en septiembre de 2022).

Un proyecto en GitHub gira en torno al concepto de repositorio de código, el cual se implementa utilizando Git. Además, un proyecto típico en GitHub incorpora las siguientes herramientas para facilitar su desarrollo de forma colaborativa:

- ***Issue tracker***, que es un sistema de gestión de tickets, o peticiones, donde los desarrolladores del proyecto pueden gestionar las tareas a abordar en el repositorio.

- ***Pull request manager***, que es el sistema de gestión de *pull requests* en Git Tal y como se ha presentado en la sección anterior, el uso de *pull requests* permite a los desarrolladores de un proyecto colaborar y discutir acerca de mejoras o correcciones en el código antes de incorporar los cambios a la versión final del proyecto.

- **Foros de discusión**, que es un sistema de gestión de foros que permite al grupo de desarrollo crear y participar discusiones acerca del producto software. En general, el tipo de discusiones que se desarrollan en los foros tienen un carácter menos técnico que las *issues*.

- **Acciones**, que son automatismos que ofrece la plataforma para gestionar determinadas tareas del proyecto. Por ejemplo, se pueden configurar acciones para ejecutar tests automáticamente o desplegar la aplicación.

- **Wiki**, que es un sistema de edición de documentación colaborativa, generalmente utilizado para ofrecer manuales, descripciones y otras indicaciones del proyecto.

Mientras que el código fuente siempre está presente en un proyecto de GitHub, el resto de herramientas se pueden activar o desactivar a conveniencia de los desarrolladores.

3. GitHub

3.5. Ejemplo práctico de Git y GitHub

En este subapartado se presentará un ejemplo práctico de clonación de un repositorio de GitHub en la máquina local y se ilustrarán los comandos principales que hay que ejecutar para registrar los cambios en local y enviarlos a GitHub.

En primer lugar, ejecutaremos el comando `clone` con la ruta para el repositorio en GitHub. El repositorio en GitHub que queremos clonar es *jlcanovas/myGitHubRepoTest*.

```
C:\git> git clone git @github.com: jlcanovas / myGitHubRepoTest.git
Cloning into 'myGitHubRepoTest'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100 % (3 / 3), done.
Receiving objects: 100 % (3 / 3), done.
remote: Total 3(delta 0), reused 0(delta 0), pack - reused 0
C:\git> cd myGitHubRepoTest
C:\git\myGitHubRepoTest> git status
On branch main
Your branch is up to date with 'origin/main'.
nothing to commit, working tree clean
```

El código anterior realiza el clone, crea un directorio llamado *myGitHubRepoTest*. Una vez dentro de dicho directorio, podemos hacer uso del comando `status` para comprobar que el repositorio está listo. Como puede observarse, la rama se llama *origin/main*, que es la nomenclatura utilizada en GitHub para indicar las ramas principales en sus repositorios.

Imaginemos que hemos hecho cambios en el fichero *README.md* del repositorio y queremos registrarlos, tanto en local como en GitHub. El siguiente fragmento de código ilustra los comandos que hay que utilizar:

```
C:\git\myGitHubRepoTest> git status
On branch main
Your branch is up to date with 'origin/main'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified: README.md
no changes added to commit(use "git add" and / or "git commit -a")
C:\git\myGitHubRepoTest> git add README.md
C:\git\myGitHubRepoTest> git commit - m "Adding changes to README.md"
[main 2 f865f5] Adding changes to README.md
 1 file changed, 1 insertion(+), 1 deletion(-)
C:\git\myGitHubRepoTest> git push
Enumerating objects: 5, done.
Counting objects: 100 % (5 / 5), done.
Writing objects: 100 % (3 / 3), 940 bytes | 940.00 KiB / s, done.
Total 3(delta 0), reused 0(delta 0), pack - reused 0
To github.com: jlcanovas / myGitHubRepoTest.git
   dc2a47a. .2 f865f5 main - > main
```

El código anterior es muy parecido al mostrado en el subapartado 2.4: realiza un `status` para ver el estado del repositorio y, a continuación, registra los cambios en el repositorio local con `add` y `commit` .

La principal diferencia radica en el uso del comando `push` , que sincroniza los cambios del repositorio local con el repositorio remoto en GitHub. Una vez realizada esta sincronización, cualquier miembro del equipo de desarrollo podría hacer un `pull` para recuperar los cambios, veamos un ejemplo:

```
C:\git myGitHubRepoTest> git status
On branch main
Your branch is up to date with 'origin/main'.
nothing to commit, working tree clean
C:\git\myGitHubRepoTest> git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100 % (5 / 5), done.
remote: Total 3(delta 0), reused 3(delta 0), pack - reused 0
Unpacking objects: 100 % (3 / 3), 921 bytes | 153.00 KiB / s, done.
From github.com: jlcanovas / myGitHubRepoTest
   2f865f5..e258016 main - > origin / main
Updating 2 f865f5..e258016
Fast - forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

Es importante recordar que la ejecución del comando `pull` anterior se realiza en una **máquina diferente** a donde se ha realizado el `push` .

Como puede observarse, el comando `status` no muestra cambios en el repositorio. Sin embargo, al ejecutar el comando `pull` , se recuperan los cambios del fichero *README.md* que previamente se habían enviado con el comando `push` desde la máquina de otro miembro del equipo de desarrollo.

De esta manera, utilizando los comandos `pull` y `push` , se utiliza GitHub como punto de sincronización entre los repositorios locales de las máquinas del equipo de desarrollo.