

BLUEJAY - TP TESTER

Índice

Introducción.....	2
¿Para qué sirve Bluejay-TP Tester?.....	2
Puesta en marcha.....	2
Modo desarrollo.....	2
Con docker.....	3
Con Electron.....	3
APIS del proyecto.....	3
Glassmatrix API.....	3
Otras APIs.....	6
Estructura proyecto angular.....	6
Páginas.....	7
Página testeo TPAs.....	7
Editar por métricas/garantías.....	7
Página métricas.....	8
Subpantalla edición de métricas.....	9
Página testeo de repositorios.....	9
Subpantalla de Ramas.....	11
Subpantalla de Pull Requests.....	11
Subpantalla de Acciones.....	12
Página de configuración.....	12
Servicios.....	13
Bluejay Service.....	13
Files Service.....	14
Github Service.....	15
Glassmatrix Service.....	16
Métodos para Operaciones de Archivos:.....	16
Métodos para Operaciones de Tokens de GitHub:.....	16
Métodos para Acciones de GitHub:.....	16
Traducciones.....	18
Electron.....	19
Swagger.....	20
Pop-ups.....	22
Ejemplos de uso.....	23
¿Cómo pruebo una métrica?.....	23
¿Cómo pruebo un TPA completo?.....	24

Introducción

Bluejay Infrastructure es una infraestructura basada en Governify que permite auditar equipos ágiles de manera sencilla. Está compuesta por un subconjunto de microservicios de Governify que pueden ser desplegados ya sea en una sola máquina o en un clúster. Bluejay accede a múltiples fuentes para recopilar información sobre los equipos de desarrollo, como GitLab, Jira, Slack, etc., y utiliza esta información para verificar si esos equipos cumplen con un Acuerdo de Prácticas de Equipo (TPA) que incluye métricas y garantías relacionadas con la metodología ágil.

¿Para qué sirve Bluejay-TP Tester?

El objetivo de este TP-Tester es, como dice su nombre, poder probar en primer lugar las métricas que componen los TPs (Team Practices). Una vez comprobadas que estas métricas funcionan correctamente, TP-Tester te permite comprobar que todo el TPA (Team Practice Agreement) funcione correctamente. Para ello, se pueden añadir, eliminar, o modificar TPAs que se encuentran directamente en Bluejay desde este TP Tester.

Puesta en marcha

Los script disponibles para la puesta en marcha son los siguientes:

```
"scripts": {
  "ng": "ng",
  "start": "concurrently \"ng serve\" \"node server.js\"",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test",
  "docker": "docker-compose up --build",
  "electron": "concurrently \"ng build --base-href ./ && electron .\" \"node server.js\""
}
```

Se podrá usar cualquiera de los scripts simplemente con “npm run nombre_script.”

Modo desarrollo

Para levantar Bluejay-TP Tester en modo desarrollo, sigue los siguientes pasos:

1. Clona el repositorio de Bluejay-TP Tester.
2. Instala las dependencias con `npm install`.
3. Levanta el proyecto con `npm start`.

Con esto bastaría ya que el proyecto utiliza *concurrently* y levanta a la vez el servidor express y angular. El servidor Express (GlassMatrix API) se levanta en el puerto 6012 y la aplicación Angular en el puerto 4200.

Con docker

Para levantar Bluejay-TP Tester con docker, sigue los siguientes pasos:

1. Clona el repositorio de Bluejay-TP Tester.
2. Instala las dependencias con ``npm install``.
3. Ejecuta ``npm run docker``.

Con esto ya tendríamos el proyecto levantado en el puerto 6011 la web angular, y el servidor express en el puerto 6012.

Con Electron

Electron es una librería que está basada en Chromium y Node.js y esto permite el desarrollo de aplicaciones de escritorio. Por tanto para levantarlo como una aplicación de escritorio será necesario únicamente seguir los siguientes pasos:

1. Clona el repositorio de Bluejay-TP Tester.
2. Instala las dependencias con ``npm install``.
3. Ejecuta ``npm run electron``.

APIS del proyecto

Glassmatrix API

Este proyecto tiene su propia API. Esta api se usa tanto para interactuar con la infraestructura de Bluejay, que permite crear TPAs, probar métricas como para realizar acciones de github en local como clonar repositorios, crear ramas, moverte de ramas, hacer commits o pushes.

Esta API se encuentra en el puerto 6012 tanto en modo desarrollo como cuando se ejecuta con docker. Esta API también utiliza Swagger, una herramienta popular para documentar APIs. Swagger ayuda a definir, construir, documentar y consumir servicios web RESTful ya que te permite incluso probar los endpoints desde una interfaz gráfica. En este caso, esta interfaz se encuentra en **localhost:6012/api** o **localhost:6012/docs** y en ella podemos ver todos los endpoints que listo y explico a continuación:

Bluejay <small>Endpoints for the Bluejay section</small>		^
POST	/glassmatrix/api/v1/tpa/save	▼
POST	/glassmatrix/api/v1/tpa/update	▼
GET	/glassmatrix/api/v1/tpa/files	▼
GET	/glassmatrix/api/v1/tpa/files/{fileName}	▼
DELETE	/glassmatrix/api/v1/tpa/files/{fileName}	▼
Github <small>Endpoints for the Github section</small>		^
POST	/glassmatrix/api/v1/github/token/save	▼
GET	/glassmatrix/api/v1/github/token/get	▼
DELETE	/glassmatrix/api/v1/github/token/delete	▼
POST	/glassmatrix/api/v1/github/cloneRepo	▼
GET	/glassmatrix/api/v1/github/listRepos	▼

1. `POST /glassmatrix/api/v1/tpa/save` permite guardar un archivo. Los parámetros `fileName` y `content` se envían en el cuerpo de la solicitud. Si hay un error al guardar el archivo, se devuelve un código de estado HTTP 500. Si el archivo se guarda correctamente, se devuelve un mensaje de éxito.
2. `POST /glassmatrix/api/v1/tpa/update` permite actualizar un archivo existente. Los parámetros `fileName` y `content` se envían en el cuerpo de la solicitud. Si el archivo existe, se borra y se crea uno nuevo con el contenido proporcionado. Si hay un error al guardar el archivo, se devuelve un código de estado HTTP 500. Si el archivo se actualiza correctamente, se devuelve un mensaje de éxito.
3. `GET /glassmatrix/api/v1/tpa/files` devuelve todos los archivos `.json` en la carpeta de activos.
4. `GET /glassmatrix/api/v1/tpa/files/:fileName` devuelve el contenido de un archivo `.json` específico. El nombre del archivo se pasa como parámetro en la ruta.
5. `DELETE /glassmatrix/api/v1/tpa/files/:fileName` permite eliminar un archivo específico. El nombre del archivo se pasa como parámetro en la ruta. Si el archivo no existe, se devuelve un código de estado HTTP 404. Si hay un error al eliminar el archivo, se devuelve un código de estado HTTP 500. Si el archivo se elimina correctamente, se devuelve un mensaje de éxito.
6. `POST /glassmatrix/api/v1/github/token/save` permite guardar un token. El token se envía en el cuerpo de la solicitud. Si el archivo que almacena el token existe, se

borra y se crea uno nuevo con el token proporcionado. Si hay un error al guardar el token, se devuelve un código de estado HTTP 500. Si el token se guarda correctamente, se devuelve un mensaje de éxito.

7. `GET /glassmatrix/api/v1/github/token/get` devuelve el token guardado. Si hay un error al leer el token, se devuelve un código de estado HTTP 500.
8. `DELETE /glassmatrix/api/v1/github/token/delete` permite eliminar el token guardado. Si el archivo que almacena el token no existe, se devuelve un código de estado HTTP 404. Si hay un error al eliminar el archivo, se devuelve un código de estado HTTP 500. Si el archivo se elimina correctamente, se devuelve un mensaje de éxito.
9. `POST /glassmatrix/api/v1/github/cloneRepo`: Esta ruta clona un repositorio de GitHub. Toma el propietario y el nombre del repositorio como parámetros del cuerpo de la solicitud, construye la URL del repositorio y clona el repositorio en un directorio local.
10. `GET /glassmatrix/api/v1/github/listRepos`: Esta ruta lista todos los repositorios clonados. Lee el directorio donde se almacenan los repositorios y devuelve una lista de nombres de directorios.
11. `GET /glassmatrix/api/v1/github/branches/:repoName`: Esta ruta lista todas las ramas de un repositorio específico. Ejecuta el comando `git branch` en el directorio del repositorio y devuelve los nombres de las ramas.
12. `DELETE /glassmatrix/api/v1/github/deleteRepo/:repoName`: Esta ruta elimina un repositorio específico. Elimina el directorio del repositorio del sistema de archivos.
13. `POST /glassmatrix/api/v1/github/createBranch/:repoName`: Esta ruta crea una nueva rama en un repositorio específico. Toma el nombre de la rama del cuerpo de la solicitud, ejecuta el comando `git checkout -b` para crear la rama y luego empuja la nueva rama al repositorio remoto.
14. `GET /glassmatrix/api/v1/github/pullCurrentBranch/:repoName`: Esta ruta tira los últimos cambios de la rama actual de un repositorio específico. Ejecuta el comando `git pull` en el directorio del repositorio.
15. `DELETE /glassmatrix/api/v1/github/deleteBranch/:repoName/:branchName`: Esta ruta elimina una rama específica de un repositorio específico. Ejecuta el comando `git branch -d` para eliminar la rama.
16. `POST /glassmatrix/api/v1/github/changeBranch/:repoName/:branchName`: Esta ruta cambia a una rama diferente en un repositorio específico. Ejecuta el comando `git checkout` para cambiar a la rama especificada.

17. `GET /glassmatrix/api/v1/github/files/:repoName``: Esta ruta lista todos los archivos en un repositorio específico. Lee el directorio del repositorio y devuelve una lista de nombres de archivos.
18. `POST /glassmatrix/api/v1/github/commit/:repoName``: Esta ruta crea un nuevo commit en un repositorio específico. Toma el contenido del archivo y el mensaje del commit del cuerpo de la solicitud, escribe el contenido en un nuevo archivo en el directorio del repositorio y luego ejecuta el comando ``git add . && git commit -m`` para crear el commit.
19. `POST /glassmatrix/api/v1/github/createFile/:repoName``: Esta ruta crea un nuevo archivo en un repositorio específico. Toma el nombre del archivo y el contenido del cuerpo de la solicitud y escribe el contenido en un nuevo archivo en el directorio del repositorio.
20. `POST /glassmatrix/api/v1/github/push/:repoName``: Esta ruta empuja los cambios a un repositorio específico. Ejecuta el comando ``git push origin`` en el directorio del repositorio.

Las últimas tres rutas (`GET /api``, `GET /docs``, y `app.listen``) no están relacionadas con la API de GitHub. Redirigen a la documentación de la API y arrancan el servidor, respectivamente.

Además de esto, la API tiene 3 redirecciones más, si se accede a `/glassmatrix/api/v1/documentation`, se podrá leer este archivo en formato pdf. Y si accedes a `/docs` te redirigirá automáticamente a la página swagger.

Otras APIs

También el proyecto interactúa con la API de github para poder realizar pushes/pulls de repositorios y comprobar así que las métricas y garantías estén funcionando correctamente. Además también interactúa con la API de bluejay para obtener los datos de computación de las métricas y obtener o crear nuevos TPAs. Esto se verá más adelante en la sección de servicios donde repaso los endpoints a los que se llaman.

Estructura proyecto angular

El proyecto utiliza la versión 13.3.11. Me decanté por usar Angular y en concreto por la versión 13 ya que esta versión Angular incluye características como el cambio de detección, la detección de zona y la renderización del lado del servidor que ayudan a mejorar el rendimiento de las aplicaciones, y es muy cómoda la interacción con distintas apis como pueden ser la api propia o la api de github.
















El proyecto se encuentra dividido en 3 grupos, componentes, páginas y servicios. A continuación detallo el funcionamiento de cada uno de ellos.

Páginas

Página testeo TPAs

Esta página permite visualizar los TPAs que ya se encuentran cargados dentro de Bluejay.

Existings TPAs

ID	Project	Class	Options
tpa-TFG-GH-antoniosc7_tpPrueba	TFG-GH-antoniosc7_tpPrueba	TFG	  
tpa-class01-GH-cs169_fa23-chips-10.5-53	class01-GH-cs169_fa23-chips-10.5-53	class01	  
tpa-TFG-GH-antoniosc7_tpPrueba2	TFG-GH-antoniosc7_tpPrueba2	TFG	  
tpa-TFG-GH-JaviFdez7_ISPP-G1-Talent	TFG-GH-JaviFdez7_ISPP-G1-Talent	TFG	  
tpa-project01	project01	class01	  

Si seleccionamos la opción de editar, se podrá editar todo el TPA o se podrá usar un editor que permite editar métrica la métrica o añadir una nueva métrica al TPA facilitando así la edición del TPA. En esta tabla también podemos encontrar la opción de simplemente ver el TPA o borrarlo (Esto lo borraría de Bluejay directamente).

Create TPA

Copy Default TPA

Editar por métricas/garantías

En esta página se permite añadir a un TPA alguna métrica o garantía de forma individual sin necesidad de tener que tocar en el json completo del TPA. Además se incluye un botón que introduce en el cuadro de texto un ejemplo de métrica o garantía que puede servir como guía.

⚠ Es importante que el nombre de la métrica o garantía sea exactamente igual que el id de la métrica o garantía.

Métricas

Crear nueva métrica Copiar métrica de ejemplo

Página métricas

Para testear métricas, en primer lugar habrá que acceder a la página de metrics tester. En primer lugar hay una sección en donde podemos realizar acciones con las métricas ya guardadas. En la parte inferior, podemos testear una nueva métrica.

Por defecto, habrá una métrica en la zona de crear nueva metrica. Esta métrica es simplemente un ejemplo y es totalmente editable o directamente se puede borrar y usar una nueva.

Crear nueva métrica

```
{
  "config": {
    "scopeManager": "http://host.docker.internal:5700/api/v1/scopes/development"
  },
  "metric": {
    "computing": "actual",
    "element": "number",
    "event": {
      "githubGQL": {
        "custom": {
          "type": "graphql",
          "title": "Get pull requests with at least one comment by member",
          "steps": {
            "0": {
              "type": "queryGetObject",
              "query": "{\n  repository(name: \"%PROJECT.github.repository%\", owner: \"%PROJECT.github.repoOwner%\") {\n    pullRequests(first: 100) {\n      pageInfo {\n        endCursor\n        hasNextPage\n      }\n      nodes {\n        bodyText\n        number\n        state\n        createdAt\n        author {\n          login\n          mergedAt\n          mergedBy {\n            login\n          }\n          comments(first: 50) {\n            nodes {\n              author {\n                login\n              }\n              bodyText\n              createdAt\n              reviews(first: 30) {\n                nodes {\n                  state\n                  createdAt\n                  bodyText\n                  author {\n                    login\n                  }\n                }\n              }\n            }\n          }\n        }\n      }\n    }\n  }"
            },
            "1": {
              "type": "objectGetSubObjects",
              "location": "data.repository.pullRequests.nodes"
            }
          }
        }
      }
    }
  }
}
```

Una vez que hemos introducido la métrica que queremos testear, hay que darle a publicar. Esto interactuará con el endpoint de Bluejay y tras obtener la computación inicial, se desbloqueará el botón de obtener cálculo. Esto tardará unos segundos y devolverá los valores de las métricas que estamos testando en ese repositorio.

Una vez que hemos comprobado la métrica (o antes si lo deseas) también habrá un botón para guardar la métrica y tenerla siempre a mano en el repositorio local con el nombre que queramos. Cuando la tengamos guardada, ya se podrá acceder a ella desde la sección superior y podremos volver ejecutarla, editarla o borrarla.

Si en la primera sección accedemos a editar métricas entraremos en una página como la inferior. Esta página tiene la opción de utilizar la hora actual (utilizará como fecha de inicio el comienzo de la hora actual, si son las 18:27, la hora de inicio será las 18:00 y la hora de final de computación serán las 18:59 del día en el que se realice). Si se guarda la métrica desde esta sección, se guardará con el periodo de cálculo utilizado.

You are executing PullRequestCerradas.json.

Save as json

Context Section

Scope Information

Project

TFG-GH-JaviFdez7_JSPP-G1-Talent

Class

TFG

Member

*

Computation Search

Type

static

Period

hourly

Initial

2024-03-14T00:00:00.000Z

From

2024-03-14T00:00:00.000Z

End

2024-03-14T23:59:59.999Z

Timezone

America/Los_Angeles

Actual hour computation

Execution Section

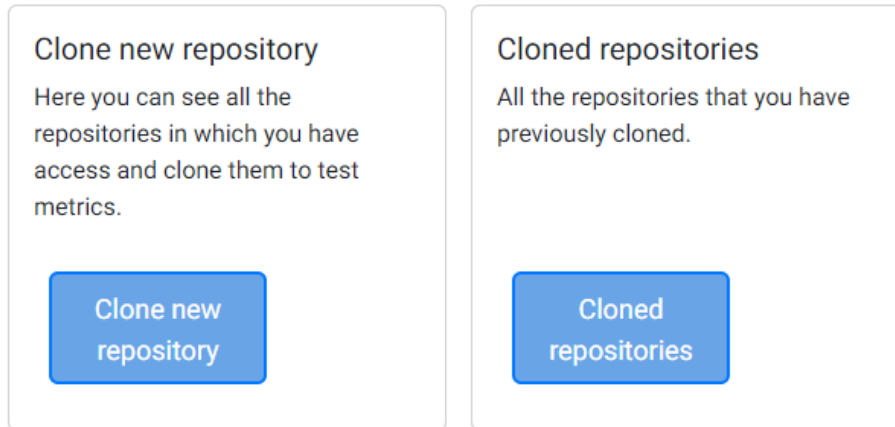
Search

Search...

```
{
  "config": {
    "scopeManager": "http://host.docker.internal:5700/api/v1/scopes/development"
  },
  "metric": {
    "computing": "actual",
    "element": "number",
    "event": {
      "githubGQL": {
        "custom": {
          "type": "graphql",
          "title": "Get closed pull requests",
          "steps": {
```

La primera de ella será la página para clonar un nuevo repositorio.

To clone a new repository you will need a valid token with access to that repository. For more information, visit: [Github token documentation](#). The token must be introduced in the clone new repository page.



En ella, al inicio hay una sección para añadir o editar el token de github y tras introducir un token válido aparecen todos los repositorios a los que tiene acceso con ese token. Además cada repositorio que aparezca listado tendrá dos opciones, verlo en github para asegurarnos de que es el repositorio que queremos clonar y otra pantalla opción es la de editar que permite clonar el repositorio en local.

Token found				
<button>View Token</button>				
Repository Name	Number of Branches	Last Update	View	Edit
Antoniiosc7	1	2024-01-10T23:20:42Z		
Antoniiosc7.github.io	1	2024-01-15T23:03:47Z		
ISSIPProject	1	2022-11-25T12:57:33Z		
JerseyDetection	2	2024-01-10T23:53:57Z		
JumpMarius	5	2024-01-15T18:20:03Z		
SnakeGamePD	1	2023-09-18T11:35:27Z		
TFG-Angular	1	2023-08-29T02:14:12Z		

Por otro lado, si en la pantalla principal de Repository Tester accedemos a repositorios clonados, listará todos los repositorios que hayamos clonado ya, y para cada uno de ellos tendremos 3 opciones, una para crear / eliminar o cambiar de rama. Otra opción para realizar pull requests y otra opción que permitirá crear archivos y realizar commits y pushes.

Subpantalla de Ramas

En esta pantalla se podrán gestionar las ramas del repositorio clonado localmente, se podrán crear, cambiar y eliminar ramas. Esto será útil para realizar commits y pushes en la subpantalla de acciones o para hacer pull requests.

tp-testbench

Branches Repository

* 321443211234	
3231	
main	

Subpantalla de Pull Requests

En esta pantalla se podrá crear una pull request, ver las pull request que estén abiertas y realizar el merge de una pull request.

Para crear una pull request se le deberá dar un título, indicar la rama que se quiere actualizar, llamado base y head será el repositorio del cual se traerán los cambios. Para cerrar la pull request se deberá introducir el número de la pull request que se quiere cerrar. Este número se podrá ver en el listado de “Open pull request” y se deberá añadir un texto para cerrarla.

tp-testbench

Create Pull Request

Base is the repository that will be updated. Changes will be added to this repository via the pull request. Head: Head is the repository containing the changes that will be added to the base.

Title:

Head:

Base:

Body:

Open Pull Requests

merge test2

Number: 3

Subpantalla de Acciones

En esta subpantalla se podrán crear archivos y realizar commits y pushes a la rama que tengamos seleccionada. Esto permitirá comprobar si alguna métrica que tenga que ver con commits o pushes funciona correctamente.

tp-testbench | Actions



[View in GitHub](#)

Available Branches

Branch Name

* 321443211234

3231

main

Current Branch:

* 321443211234

Change Branch

Create New File

New file name

New file content

Create File

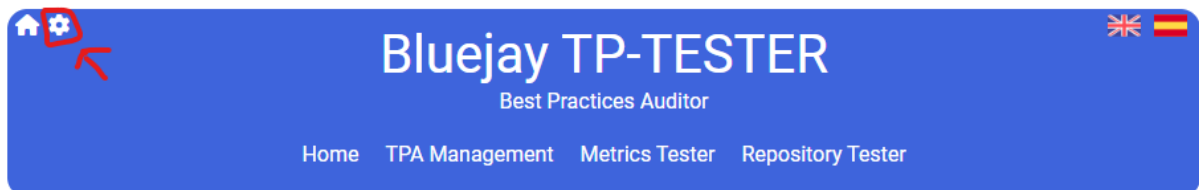
Create New Commit

Commit message

Create Commit

Página de configuración

En la parte superior izquierda del header se puede encontrar un icono de configuración. Si accedemos a este icono entraremos en la página de configuración.

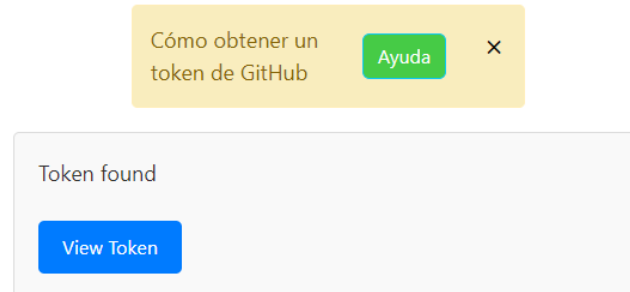


Esta página actualmente está compuesta por 4 secciones que ayudan a entender el funcionamiento de la página.

La primera de ellas, es Dockers activos, aquí se pueden visualizar los contenedores de Bluejay que se encuentran levantados pudiendo verse así fácilmente si hay algún error en la página porque estos no estén conectados. Estos contenedores solo se podrán ver si se ejecuta la página en modo desarrollador ya que si se hace a través de docker no podrá verlos acceder a la api de docker de los otros contenedores.

La segunda sección es Github Token. Este componente también se encuentra en la página de clonados de repositorios, pero también está en esta página de configuración para que sea más accesible. Además hay un warning que al pulsar sobre él abre un pop-up con información de como crear un token correcto de github.

Github Token



La tercera sección es Swagger, en ella se explica en qué consiste Swagger y tiene un botón que te lleva a la página de Swagger donde se pueden encontrar todos los endpoints disponibles.

Y la última sección es Documentación y en ella se puede leer este pdf.

Al lado del icono de configuración, también podemos ver el icono de una casa, al pulsar sobre él, si estamos en local con los contenedores de Bluejay levantados, se abrirá en un pop-up la página principal de Bluejay.

Servicios

Bluejay Service

El servicio Bluejay Service en Angular está diseñado para interactuar con un servidor de backend que proporciona una API relacionada con acuerdos (agreements) y cálculos computacionales. Veamos cada método del servicio y lo que hace:

1. **createTpa(tpaContent: string):** Este método toma el contenido de un Acuerdo de Procesamiento de Terceros (TPA) en forma de cadena JSON, lo convierte en un objeto JavaScript utilizando `JSON.parse`, y luego lo envía al servidor a través de una solicitud HTTP POST a la URL definida en `this.url`. El servidor espera recibir un nuevo TPA y devuelve una respuesta que se emite como un Observable.
2. **getTps():** Este método realiza una solicitud HTTP GET al servidor para obtener una lista de todos los Acuerdos de Procesamiento de Terceros (TPA) disponibles. La respuesta del servidor se emite como un Observable.
3. **getTpa(id: string):** Este método realiza una solicitud GET al servidor para obtener un TPA específico identificado por su ID. La URL para la solicitud se construye concatenando el ID proporcionado al final de la URL base definida en `this.url`. La respuesta del servidor se emite como un Observable.

4. **deleteTpa(id: string):** Este método realiza una solicitud DELETE al servidor para eliminar un TPA específico identificado por su ID. La URL para la solicitud se construye de manera similar al método getTpa(). La respuesta del servidor se emite como un Observable.
5. **postComputation(data: any):** Este método realiza una solicitud POST al servidor para enviar datos de computación. La URL para la solicitud se define explícitamente como `http://localhost:5500/api/v2/computations`. Los datos de computación se envían en el cuerpo de la solicitud y la respuesta del servidor se emite como un Observable.
6. **getComputation(computationUrl: string):** Este método realiza una solicitud GET al servidor para obtener los resultados de una computación específica. La URL para la solicitud se pasa como parámetro y la respuesta del servidor se emite como un Observable.

En resumen, el servicio BluejayService proporciona métodos para interactuar con los endpoints de Bluejay que gestionan las llamadas permitiendo la creación, obtención, eliminación y consulta de datos relacionados con los acuerdos, ya sean TPAs enteros o métricas. Las solicitudes HTTP se realizan utilizando el servicio HttpClient de Angular, y las respuestas del servidor se emiten como Observables para su consumo en la aplicación Angular.

Files Service

Este servicio, proporciona métodos para acceder a archivos JSON almacenados localmente en la aplicación. Estos archivos JSON pueden contener datos de configuración o ejemplos.

1. **getDefaultTPA():** Este método devuelve el contenido del archivo defaultTPA.json ubicado en la carpeta assets de la aplicación Angular. Se utiliza la opción `responseType: 'text'` para indicar que el contenido del archivo debe ser tratado como texto.
2. **getBasicMetric():** Este método devuelve el contenido del archivo basicMetric.json ubicado en la carpeta assets del proyecto.
3. **getSavedMetric(fileName: string):** Este método toma el nombre de un archivo como argumento y devuelve el contenido del archivo correspondiente ubicado en la carpeta assets/savedMetrics del proyecto.
4. **getExampleMetric():** Este método devuelve el contenido del archivo exampleMetric.json ubicado en la carpeta assets/examples del proyecto.
5. **getExampleGuarantee():** Este método devuelve el contenido del archivo exampleGuarantee.json ubicado en la carpeta assets/examples del proyecto.

En resumen, este servicio proporciona una manera conveniente de cargar archivos JSON locales en la aplicación Angular utilizando el servicio HttpClient.

Github Service

En este servicio, utilizo la API pública de GitHub para realizar una serie de operaciones relacionadas con los repositorios y las solicitudes de extracción. Aquí hay un resumen de las operaciones que realizo:

1. **listBranchesForRepo**(owner: string, repo: string): Esta función obtiene la lista de ramas de un repositorio específico. Utilizo la API de GitHub haciendo una solicitud GET a la URL correspondiente del repositorio.
2. **getUserName**(token: string): Esta función obtiene el nombre de usuario asociado a un token de autenticación. Utilizo la API de GitHub haciendo una solicitud GET a la URL del usuario autenticado.
3. **listRepos**(token: string): Esta función obtiene la lista de repositorios públicos del usuario autenticado. Utilizo la API de GitHub haciendo una solicitud GET a la URL de los repositorios del usuario autenticado.
4. **getLatestCommitSha**(owner: string, repo: string): Esta función obtiene el SHA del último commit en un repositorio específico. Utilizo la API de GitHub a través de la librería Octokit para obtener la lista de commits y luego extraer el SHA del último commit.
5. **getRepoInfo**(owner: string, repo: string): Esta función obtiene información detallada sobre un repositorio específico. Utilizo la API de GitHub haciendo una solicitud GET a la URL del repositorio.
6. **createBranch**(owner: string, repo: string, branchName: string, ref: string): Esta función crea una nueva rama en un repositorio específico. Utilizo la API de GitHub a través de la librería Octokit para crear la nueva rama.
7. **createFile**(owner: string, repo: string, path: string, message: string, content: string, branch: string): Esta función crea un nuevo archivo en un repositorio específico. Utilizo la API de GitHub a través de la librería Octokit para crear el archivo.
8. **createPullRequest**(token: string, owner: string, repo: string, prTitle: string, prHead: string, prBase: string, prBody: string): Esta función crea una nueva solicitud de extracción (pull request) en un repositorio específico. Utilizo la API de GitHub haciendo una solicitud POST a la URL de las solicitudes de extracción del repositorio.
9. **getOpenPullRequests**(token: string, owner: string, repo: string): Esta función obtiene la lista de solicitudes de extracción abiertas en un repositorio específico.

Utilizo la API de GitHub haciendo una solicitud GET a la URL de las solicitudes de extracción abiertas del repositorio.

10. **mergePullRequest**(token: string, owner: string, repo: string, mergePrNumber: number, mergeCommitMessage: string): Esta función fusiona una solicitud de extracción específica en un repositorio específico. Utilizo la API de GitHub haciendo una solicitud PUT a la URL de la solicitud de extracción a fusionar.
11. **getBranches**(token: string, owner: string, repo: string): Esta función obtiene la lista de ramas de un repositorio específico. Utilizo la API de GitHub haciendo una solicitud GET a la URL correspondiente del repositorio.

En resumen, estoy utilizando la API pública de GitHub para realizar diversas operaciones relacionadas con repositorios y solicitudes de extracción, aprovechando las funcionalidades proporcionadas por la API para interactuar con los recursos de GitHub.

Glassmatrix Service

El servicio GlassmatrixService en Angular está diseñado para interactuar con un servidor de backend propio que he explicado anteriormente en el apartado de Glassmatrix API. En resumen realiza operaciones relacionadas con archivos, tokens de GitHub y acciones específicas de GitHub, como clonar repositorios, manipular ramas, crear y editar archivos, y realizar operaciones de commit y push.

Métodos para Operaciones de Archivos:

1. **saveToJson**(fileName: string, data: any): Envía una solicitud POST al servidor para guardar datos en un archivo JSON especificado.
2. **deleteFile**(fileName: string): Envía una solicitud DELETE al servidor para eliminar un archivo especificado.
3. **loadFiles**(): Envía una solicitud GET al servidor para obtener una lista de nombres de archivos disponibles.
4. **updateFile**(fileName: string, data: any): Envía una solicitud POST al servidor para actualizar un archivo JSON existente.

Métodos para Operaciones de Tokens de GitHub:

1. **getToken**(): Envía una solicitud GET al servidor para obtener un token de GitHub.
2. **saveToken**(token: string): Envía una solicitud POST al servidor para guardar un token de GitHub.
3. **deleteToken**(): Envía una solicitud DELETE al servidor para eliminar un token de GitHub.

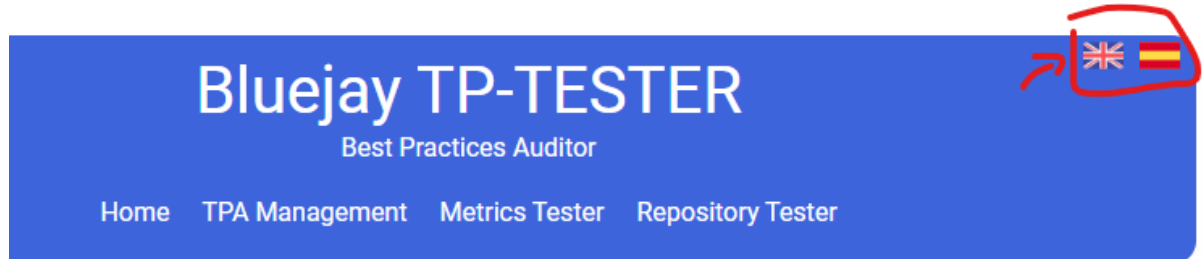
Métodos para Acciones de GitHub:

1. **cloneRepo**(owner: string, repoName: string): Envía una solicitud POST al servidor para clonar un repositorio de GitHub.
2. **listRepos**(): Envía una solicitud GET al servidor para obtener una lista de repositorios de GitHub disponibles.
3. **deleteRepo**(repo: string): Envía una solicitud DELETE al servidor para eliminar un repositorio de GitHub.
4. Métodos relacionados con operaciones de ramas, commits y manipulación de archivos en un repositorio de GitHub.

En resumen, el servicio GlassmatrixService proporciona una interfaz para interactuar con diferentes aspectos de un servidor de backend relacionado con operaciones de archivos y acciones específicas de GitHub. Esto permite a la aplicación Angular realizar diversas operaciones, como guardar y eliminar archivos, manipular repositorios de GitHub y gestionar tokens de autenticación de GitHub. Las solicitudes HTTP se realizan utilizando el servicio HttpClient de Angular, y las respuestas del servidor se emiten como Observables para su consumo en la aplicación Angular.

Traducciones

Gracias a la librería ngx-translate <https://github.com/ngx-translate/core>. Por defecto, TP Tester, estará en el idioma del navegador. Pero esto también se puede cambiar en la parte superior derecha ya que ahí podremos cambiar entre español e inglés de una manera rápida.



Esta librería permite además añadir traducciones de una forma sencilla ya que bastaría con traducir el JSON de un idioma a otro y no habría que modificar ningún otro tipo de fichero.

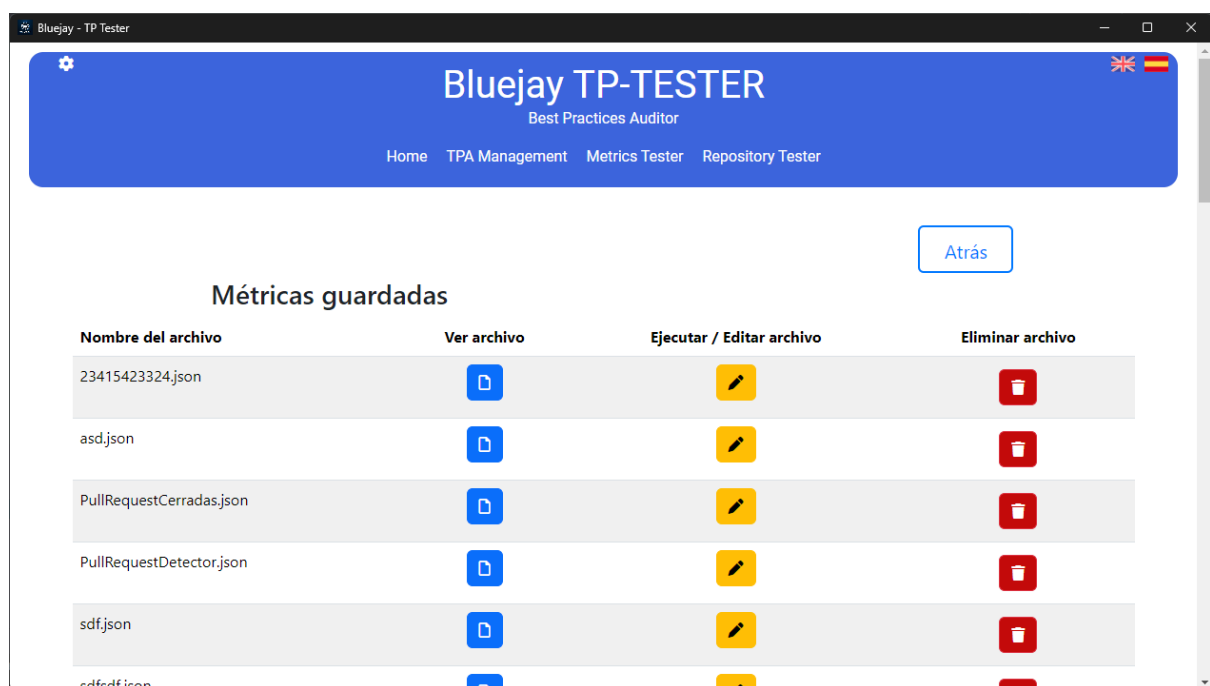
Este es un fragmento de cómo funciona el .json de la traducción a español:

```
"METRICS_LOADER": {
  "TITULO1": "Métricas guardadas",
  "TITULO2": "Crear nueva métrica",
  "MESSAGE_TEXT": "No hay métricas guardadas",
  "FILE_NAME": "Nombre del archivo",
  "VIEW_FILE": "Ver archivo",
  "EXECUTE_EDIT_FILE": "Ejecutar / Editar archivo",
  "DELETE_FILE": "Eliminar archivo",
  "VIEWER": {
    "VIEWING": "fileName."
  }
}
```

Electron

Como he comentado en el punto de puesta en marcha, el proyecto también se puede ejecutar con Electron. Electron es una librería que permite crear aplicaciones de escritorio compatibles con Windows, Linux o Mac.

La aplicación de escritorio será exactamente igual con la única diferencia que no estará el icono de la casa en la parte superior izquierda. Esto se debe a que este botón te abre un pop up con la página inicial de Bluejay, pero esta no se puede abrir mediante electron ya que necesita autenticación. Esta autenticación se realiza a través de un alert y electron, por motivos de seguridad no permite que se abran, por lo que al no poderse autenticar, la página de Bluejay no cargaba.



Swagger

Como también he comentado anteriormente, para la API de Glassmatrix he utilizado Swagger junto a Express js para crearla. Swagger es una herramienta muy útil ya que permite que el servidor Express quede bien estructurado y se puedan hacer pruebas sin necesidad de otros servicios como podrían ser Postman.

```
* @swagger
* /glassmatrix/api/v1/github/pullCurrentBranch/{repoName}:
* get:
*   tags: [Github]
*   description: Use to pull the current branch of a specific repository
*   parameters:
*     - name: repoName
*       description: Name of the repository
*       in: path
*       required: true
*       type: string
*   responses:
*     '200':
*       description: A successful response
*/
Antonio Saborido
app.get(apiName + '/github/pullCurrentBranch/:repoName', async (req : Request<P, ResBody, ReqBody>, res) => {
  const { repoName } = req.params;
  const repoPath : string = path.join(__dirname, 'assets', 'repositories', repoName);

  try {
    const { stdout, stderr } = await exec( argv: 'git pull', { cwd: repoPath });
    if (stderr) {
      console.error('Error pulling current branch:', stderr);
      res.status( code: 500 ).send( body: 'Error pulling current branch: ' + stderr );
    } else {
      res.json( body: { message: 'Pulled current branch successfully' } );
    }
  } catch (err) {
    console.error('Error executing git command:', err);
    res.status( code: 500 ).send( body: 'Error executing git command: ' + err.message );
  }
});
```

Como se puede ver en la imagen, para usar Swagger basta con añadir bloques como el verde indicando que es un bloque de Swagger (con @swagger).

El apartado tags, se utiliza para agrupar los endpoints, por ejemplo, en mi API, distingo dos bloques, aquellos que trabajan con métodos de Github y aquellos que interactúan con los endpoints de Bluejay.

Además, Swagger también te permite definir cómo tienen que ser el tipo de las variables que recibe cada endpoint para así permitirte probar con datos adecuados en la interfaz gráfica.

Esto se puede ver accediendo al puerto 6012 y si busco el mismo endpoint de la captura anterior, se vería así:

GET /glassmatrix/api/v1/github/pullCurrentBranch/{repoName}

Use to pull the current branch of a specific repository

Parameters Cancel

Name	Description
repoName * required string (path)	Name of the repository

tp-testbench

Execute Clear

Responses Response content type: application/json

Curl

```
curl -X 'GET' \
'http://tfg.antoniosaborido.es:6012/glassmatrix/api/v1/github/pullCurrentBranch/tp-testbench' \
-H 'accept: application/json'
```

Request URL

```
http://tfg.antoniosaborido.es:6012/glassmatrix/api/v1/github/pullCurrentBranch/tp-testbench
```

Server response

Code	Details
200	Response body

```
{
  "message": "Pulled current branch successfully"
}
```

Download

Como se puede ver, he introducido el nombre de un repositorio, y esta llamada ha realizado un pull de esa rama en el repositorio local y ha devuelto un código 200 ya que ha salido todo correcto.

Y como he comentado anteriormente, desde aquí mismo se podría ejecutar el código ya que realiza las llamadas realmente. Por ejemplo, en la siguiente captura intento realizar un pull de una rama que no existe y que no tengo clonada, por lo que da un error 500.

GET /glassmatrix/api/v1/github/pullCurrentBranch/{repoName}

Use to pull the current branch of a specific repository

Parameters Cancel

Name	Description
repoName * required string (path)	Name of the repository

rama_no_existente

Execute Clear

Responses Response content type: application/json

Curl

```
curl -X 'GET' \
'http://tfg.antoniosaborido.es:6012/glassmatrix/api/v1/github/pullCurrentBranch/rama_no_existente' \
-H 'accept: application/json'
```

Request URL

```
http://tfg.antoniosaborido.es:6012/glassmatrix/api/v1/github/pullCurrentBranch/rama_no_existente
```

Server response

Code	Details
500 Undocumented	Error: Internal Server Error

Response body

```
Error executing git command: spawn C:\Windows\system32\cmd.exe ENOENT
```

Download

Pop-ups

A lo largo de la página se pueden encontrar distintos botones que abren un pop-up o un dialog dentro de la página ofreciendo más información sobre esa sección.

Por ejemplo, en la página de edición de una métrica, se puede encontrar el botón de tutorial, al pulsar sobre este botón se abrirá una ventana dentro de esa misma página en la cuál se explican los tipos de métricas que se pueden crear con ejemplos de cada tipo.

Estás ejecutando 23415423324.json.

Sección de Contexto

Información de la métrica

Proyecto

TFG-GH-JaviFdez7_ISPP-G1-Talent

Clase

TFG

Miembro

*

Período de cálculo

Tipo

static

Periodo

hourly

Inicial

2024-03-14T00:00:00.000Z

Desde

2024-03-14T00:00:00.000Z

Hasta

2024-03-14T23:59:59.999Z

Zona horaria

America/Los_Angeles

Cálculo de hora actual

Guardar como json

Tutorial

Sección de Ejecución

Buscar

Buscar...

Bluejay TP-TESTER

Best Practices Auditor

Funcionamiento de las métricas

Cerrar

Formato y funcionamiento de la métrica

config: Contiene la configuración de la métrica. En este caso, "scopeManager" es la URL de bluejay y esto nunca deberá editarse.

metric: Contiene la definición de la métrica.

computing: Define cómo se calcula la métrica. En este caso, "actual" significa que se calcula en tiempo real.

element: Define el tipo de elemento que se mide. En este caso, "number" significa que se mide un número.

event: Define el evento que dispara el cálculo de la métrica. En este caso, se utiliza una consulta personalizada de GraphQL a la API de GitHub. Esto puede ser modificable como se ve en el apartado inferior de GraphQL custom query

scope: Define el alcance de la métrica. En este caso, se mide a nivel de tpa, y para todos los miembros (*).

window: Define el período de tiempo para el cual se calcula la métrica. En este caso, se calcula por hora para un día específico.

Metrica de ejemplo:

```
{  "config": {    "scopeManager": "http://host.docker.internal:5700/api/v1/scopes/development"  },  "metric": {    "computing": "actual",    "element": "number",    "event": {      "githubGQL": {
```

Ejemplos de uso

¿Cómo pruebo una métrica?

Para probar una nueva métrica en primer lugar deberemos acceder a la pantalla de Metrics Tester. En ella veremos las que están previamente guardadas y en la sección de Crear Nueva métrica tenemos una métrica de ejemplo. Podemos ir editando sobre esta métrica de ejemplo los campos que queramos, o si ya tenemos una métrica de ejemplo que se nos ajuste más, podemos eliminarla directamente y pegar la métrica que queramos y trabajar sobre ella directamente.

Una vez tengamos ya la lista que queremos probar, podemos guardarla dándole un nombre al archivo .json que se creará para tenerla guardada por si hubiera algún problema y ya posteriormente podríamos editar sobre este archivo.

Una vez guardada, le podemos dar al botón de “post” o “publicar”. Esto tardará unos segundos ya que tiene que interactuar con Bluejay, cuando termine el spinner de cargando aparecerá un mensaje en el textarea inferior.

Publicar

Obtener Cálculo

Buscar en la respuesta...

Buscar en la respuesta

```
{  
  "code": 200,  
  "message": "OK",  
  "computation": "/api/v2/computations/6ed8959da3976867"  
}
```

Si el código es 200, significa que la métrica es válida y se nos desbloqueará el botón de obtener cálculo. Una vez le demos ya realizará los cálculos para el periodo seleccionado y devolverá algo así:

Publicar

Obtener Cálculo

Buscar en la respuesta...

Buscar en la respuesta

```
{
  "code": 200,
  "message": "OK",
  "computations": [
    {
      "scope": {
        "project": "TFG-GH-JaviFdez7_ISPP-G1-Talent",
        "class": "TFG",
        "member": "Manuel_OteroBarbasán"
      },
      "period": {
        "from": "2024-03-14T00:00:00.000Z",
        "to": "2024-03-14T01:00:00.000Z"
      },
      "evidences": [],
      "value": 0
    },
    {
      "scope": {
        "project": "TFG-GH-JaviFdez7_ISPP-G1-Talent",
        "class": "TFG",
```

Si el “value” es 1, significará que ha detectado esa métrica en el periodo seleccionado, y si es 0 pues no habrá encontrado esa métrica para ese periodo.

¿Cómo pruebo un TPA completo?

Para ello deberemos acceder a la página de TPA management. En la sección de TPAs existentes mostrará aquellos que ya se encuentran cargados en Bluejay. Y en la sección de crear TPA podremos crear uno nuevo. En esta sección también contamos con el botón de copiar un TPA por defecto, y al igual que hacíamos con las métricas, podemos editarlo en ese textarea propio.

Una vez que lo creemos este TPA se subirá a Bluejay de forma automática. En esta página de [documentación de bluejay](#) se puede encontrar más información sobre la estructura de los TPA.

Una vez lo hayamos creado seguiremos pudiendo modificarlo. Para ello en donde aparecen todos los TPAs cargados en Bluejay podremos darle al botón de editar y nos dará dos opciones, si editar el json del tpa completo o editarlo por métricas o garantías. No solo podremos editar las métricas o garantías ya creadas sino que también se podrán crear nuevas a través de esta página.

Cuando hayamos realizado todas las modificaciones oportunas y lo guardemos, lo que hará realmente la página será un delete del tpa que estaba ya creado y volverá a postearlo con la nueva información.