**Version Control Software Overview**

****Version control is a method of tracking and managing changes to a project. Prominent terms mentioned include Git, a powerful and widely used VCS tool; Repository (repo), which is essentially a project folder managed by Git; and GitHub, a popular web-based platform for hosting Git repositories and facilitating teamwork. It's a common misconception, but it's important to remember that Git (the tool) and GitHub (the hosting service) are distinct entities.

**Git Overview:**

[Git](https://git-scm.com/) is a distributed version-control system that tracks changes to files in a folder (or repo) over time. This allows the user to experiment with their project and revert back to previous versions at any time. And when the user is ready, they can save new changes adding to the folders history

**Repositories:**

A repository (or “repo”) is a storage location for all the files related to a project such as models, charts, datasets, and other supporting files. Along with folder called “.Git” which is where Gits configurations are held.

There are two main types of repositories:

* Local Repository: This is stored on a user's own computer. Changes are made, saved, and tracked locally.
* Remote Repository: This exists on a server or hosting platform and is used to share the project with others and back it up online.

These two types of repositories work together. Users typically make changes in the local repository, and when those changes are ready, they are pushed to the remote repository so others can see or work with the latest version.

Remote version control hosts are platforms that store project repositories online. These remote hosts allow pushing and pulling of updates between local and remote versions of a project allowing for efficient workflows and development tracking.

There are many benefits such as:

* Secure backups of the entire project history
* Ability to revert to any previous version
* Access for the whole team, from anywhere

GitHub along with sites like Azure DevOps and GitLab are popular examples of remote version control hosts

**A screenshot of a phone

AI-generated content may be incorrect.Development Cycle**

**Basic Workflow**

In a simple workflow a developer will:

* **Pull** before modifying any code to ensure they are working on the latest version of the repository
* The user will then make their changes and **commit** them to the local repository with a descriptive message about their changes
* **Push** their changes from the local to remote repository to ensure the rest of their team can access their latest changes with a simple pull of the remote repository. (which happens by default in Simio using our commit and push button)

**Advanced Development Habits**

The basic workflow is by no means the extent of Git’s suite of tools. There are a ton of helpful actions a user can use to aid in their development. An example of this is **Branching,** which is an essential tool for large scale projects and will be covered in depth in its own section.

**Git Terms and Tools**

**Git Core Functions:**

[Git Pull](https://git-scm.com/docs/git-pull)

Git Pull downloads the latest version of a project from a central shared location (remote repository, such as GitHub) to the local project folder. It updates the local repository with any new changes made by others, ensuring that the local copy is in sync with the most current version of the project.

[Git Commit](https://git-scm.com/docs/git-commit)

Git commit is the action that stores the changes of all the files in use into what is intuitively also called a commit. Think of a commit as saving a version of an entire project folder (including all charts, models, and anything else not explicitly ignored). When changes are made such as updating process logic or adding a resource, or anything else, Git commit captures a snapshot of everything in that moment. Which can be reverted to, shared or just stored using other available Git functions. Additionally, each of these commits include a message where the user can describe the changes they made such as “Added new process logic to Model1” or “Fixed resource assignment in Station3.” These messages help the developer, and the rest of their team understand what has been done and can see a road map of changes over their development process.

[Git Push](https://git-scm.com/docs/git-push)

Git push is the opposite of Git pull. After changes have been committed locally, Git push uploads those commits to the central shared location (the remote repository). This action makes the new changes available to other collaborators and keeps the remote repository up to date.

**Branching with Git:**

**Overview and Functionality of Branching**

Branching is an essential tool for larger development. It is what allows developers to be experimental without breaking their projects. Think about some of the software, there are usually releases/versions of software. It is tempting to think the development behind the scenes was simply to get from version A to version B; however, it is often not that linear. These releases are the tip of the iceberg when it comes to development. That's where branching comes in. Each release is a commit on the main branch – a snapshot of the project when it is stable and ready to be published. When a developer wants to tinker with the project and makes some changes, they can check it out into a development branch where they can commit and save their project freely without corrupting the main release branch. Then when it is stable and tested, they can then push their new features/bug fixes to the main release branch as the newest version. However, if at any point the development is going in the wrong direction they can simply restart and go back to the main.

**Branching Workflow**

A screenshot of a computer

AI-generated content may be incorrect.When getting started with branching it is assumed there is already a main. This is because a main branch is created automatically when first initializing a repository as the default branch. So, to begin the user must create a development (dev) branch.

In best practice a developer will:

* Create a branch off the main branch either from its head (the most recent commit) or from a specific commit if the head is corrupted and the developer wants to go back to a previous version.
* Treat the branch exactly as if the user was working off the main in a basic workflow with the caveat of sharing. In the case of collaboration/sharing on a branch not only does the repo need to be cloned but the branch will also need to be “checked out” as by default Git only clones the Main branch.
* When the development branch is tested and ready to be released as part of the main branch, the developer will merge the dev branch with the main having the final commit on the dev branch now being the head of the main branch

*\*As a note the not all branches have to be merged with the main branch, and it is dependent on the project and what works best for development needs.*

Merge Conflicts

Git can be useful when working with others, streamlining file sharing and collaboration. However, it is not a replacement for communication. Lack of communication between developers can cause issues such as merge conflicts where two people have made changes simultaneously and sync the changes together if they are affecting the same file. This can be overwritten in a code editor however it can be challenging to resolve and there is a definite learning curve on handling such errors and best practice is to avoid them through communication with the development team.

.Gitignore

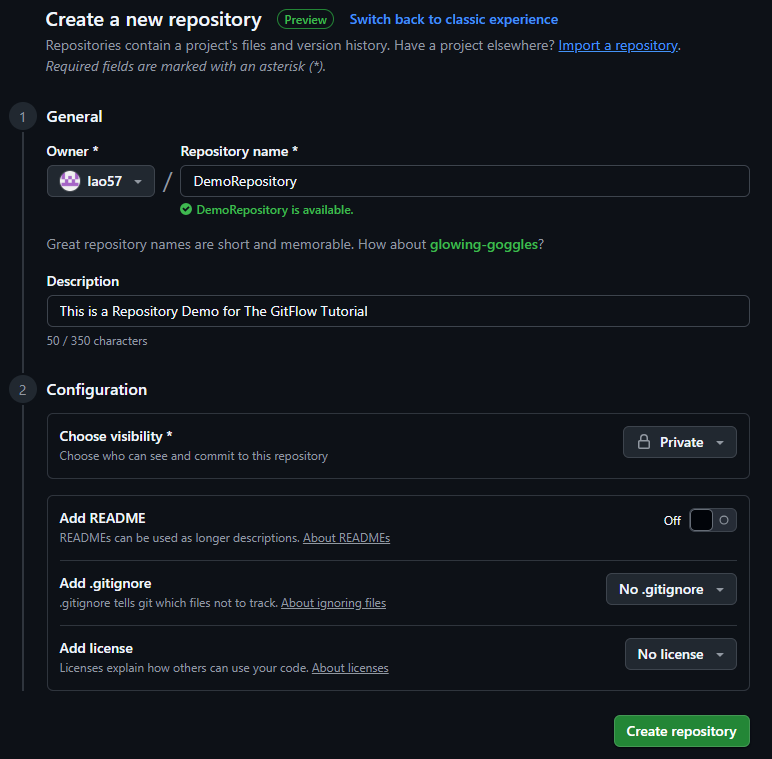
The “.Gitignore” file stops Git from tracking certain files or file types, stopping them from being pushed to the remote repository. It can be used to avoid uploading tables, model trace or other files that are not necessary to share with other users.

PAT’s

Personal Access Tokens are a way to keep work private and protected along with simplifying the sharing process. The beauty of PAT’s is their scope customization. Users can give as much or as little access to their work as appropriate, with useful settings from specific repository access to custom expiration. This makes managing private data easy.

**Getting Started with GitFlow**

**Creating Repository Locally:**



In GitHub navigate to the CREATE repository and apply the following settings for a clean start to GitFlow. In this step, choose whether to make the repository public or private and give it any valid name or description. If a license/Readme is required, it is recommended to follow this whole procedure until the repository is initialized locally then add those files through GitHub and pull all changes using built in pull feature of Add-In.

A screenshot of a computer

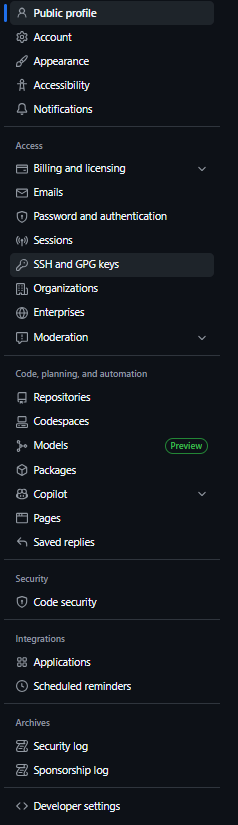
AI-generated content may be incorrect.

Now that the repository is created be sure to copy the https URL as it will be needed later to connect to a local repository.

**Creating a PAT**

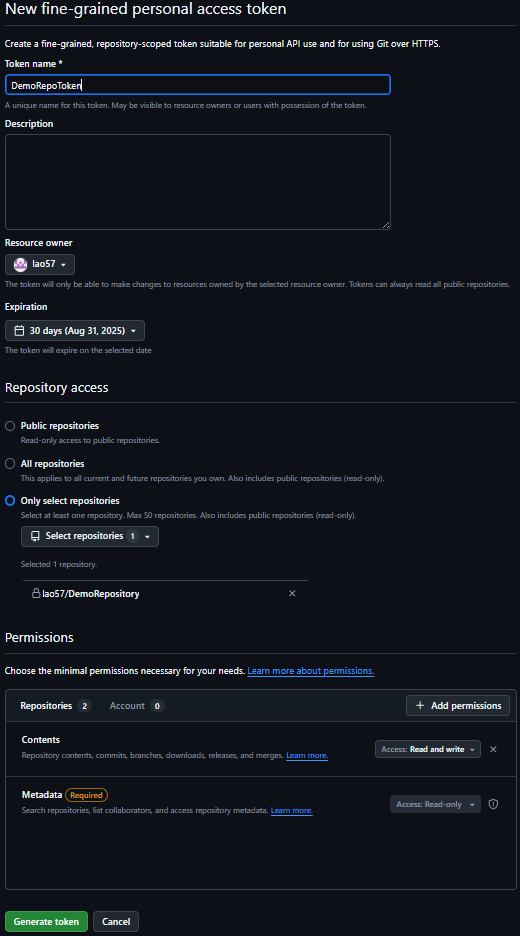
Now navigate by pressing on the profile Icon in the top right of this screen. A column should appear with an option for settings. Once clicked a menu on the left side of the screen should appear, and after scrolling to the bottom there will be option for Developer settings.

A screenshot of a computer menu

AI-generated content may be incorrect. ****

Usually, best practice in these settings is to create a fine-grained PAT as this will allow the user more control. Be sure to select the correct repository to grant access to. Then for permission select Contents [read and write] and then metadata (by default will be) [read only]. It should look like the picture below when done. Now generate but be sure not to close the following screen because the token must be saved as it will be needed later.

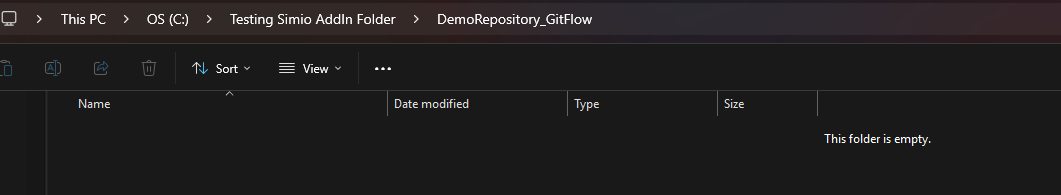
**Example: github\_pat\_1234…**

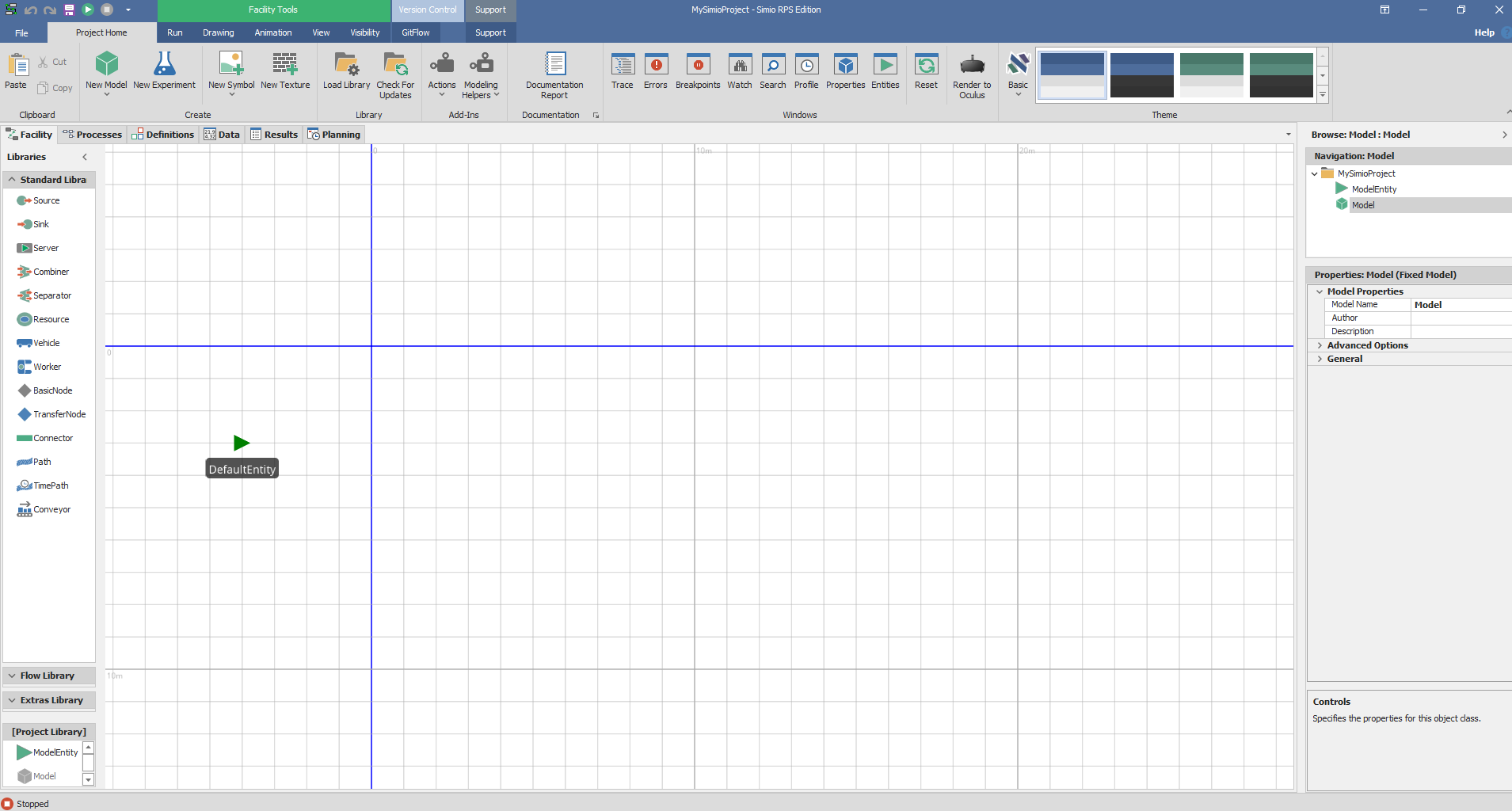
****

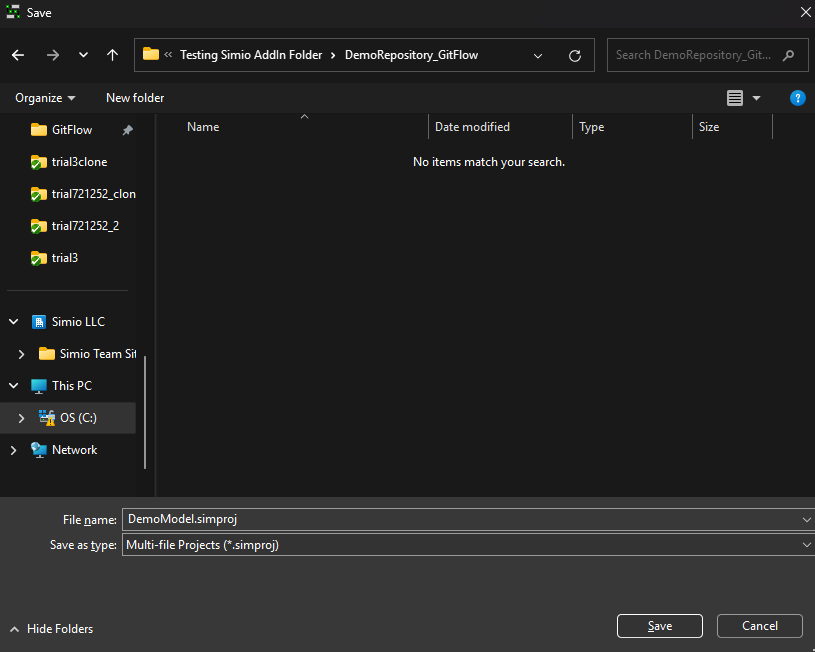
**Creating Repository Locally:**

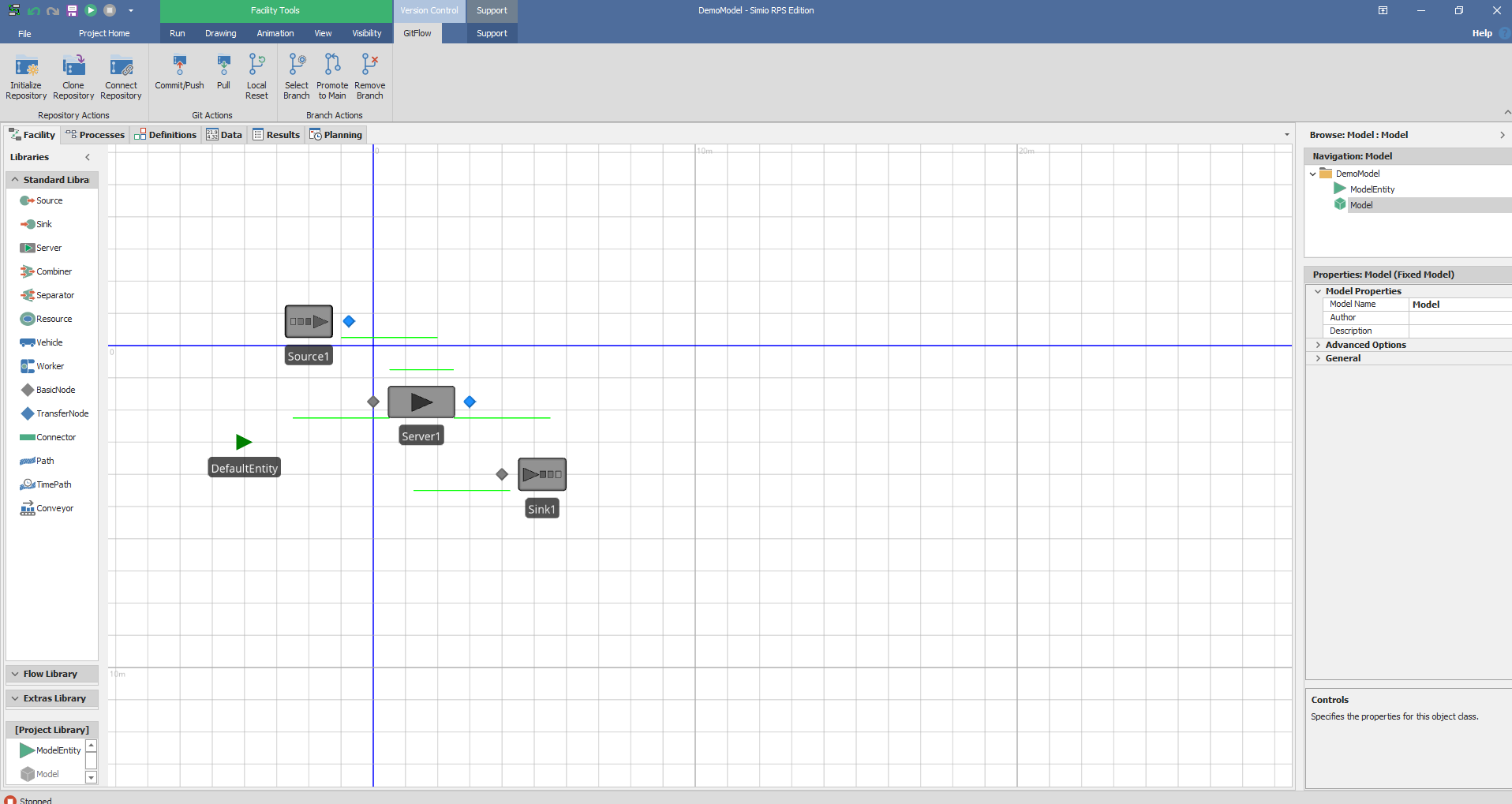
The next step to getting started is to create a folder to store the repository in. There are two crucial details to this.

1. It must not be contained in a OneDrive or any other cloud storage service. It is recommended to be in C:/ or a near direct child folder.
2. It must be an EMPTY folder.



In Simio make sure there is a ribbon for GitFlow otherwise the add-in is not properly installed.

Now open the project to be stored in the new repository and use *save as* to save it to the folder that made in the previous step. One crucial step here is before pressing save, be sure to save as a “.simproj” file as they work much better with version control.

Once this is done, the project is ready to be initialized. Press the Initialize Repository button. Proceed to filling out the form with the URL and PAT from previous steps. To fill in path it is recommended to press the ellipse and then select on the folder previously made which should at this point look like the image below with nothing but Simio files and any other files desired to be tracked by this repository.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

**Congratulations,** the project is now set up for version Control and the GitHub should look something like this. At this point feel free to add a README or License, just be sure to pull in Simio on all branches in use.