DEV Team Knowledge 2
- Our Azure Board 4
  - User Stories Effort Evaluation 6
- Internal Agile Implementation 7
  - Sprint Meeting Checklist 9
- General Glossary and Acronyms 11
- Quickstarts 13
  - Refresher Course for new members 14
  - Python Environments 19
  - Local Docker database 20
    - Postgres 21
    - Mongodump 22
  - How to install a flask service on IIS 24
  - VSCode tips 26
  - Git tips 27
- Software Development Practices 28
  - Which Dev Flow to use 29
    - Git Flow 31
    - GitHub Flow 34
  - Docker Usage 35
    - Glossary 38
  - Architecturing projects 39
    - 1 - End User 40
    - 2 - User Stories 41
    - 2.1 - Create user rights table 42
    - 3 - System Features 43
    - 4 - Software Components 44
    - 5 - The APIs 45
    - 6 - The Software 46
  - Toolchain Setup 47
  - Deployement : What to do 48
  - Versioning your projects 49
  - Code re-usability 51
  - Indexes on database 53
  - Conventionnal naming in project 54
    - Back-end Convention 55
    - Front-end Convention 56
  - How to win at Microservices 57

DEV Team Knowledge

Where am I ?

You are in the DEV Teams confluence spaces. Here you will find all the documentation we wrote during our existence.

Even on separate projects 🖐 we must work as a team 👊. This is especially true when we are so few developers scattered across so various subjects and problems. Thus we must **bring together our collective knowledge** in a way that can **benefit everybody** and make us **save time and energy !**

What can I find here ?

Literally every documentation we wrote or code we produced will be stored in one of those places. Each space stores some type of documentation you might want to store. In case of doubt, Ask !

Please bookmark this page

DEV Team Knowledge

- 📐 Technological and technical knowledge
- 🏹 How we use Agile to organize our work
- General NovAliX molecular knowledge useful for us DEVs
- 💡 Tips for our day2day usage of various software at NovAliX

> How we develop : Software Development Practices
>
> How we go from dev to prod : Which Dev Flow to use

DEV Team Infrastructure

- ⛓ Service Mesh on various environments (DEV, STAGING, QA, PRODUCTION)
- 💡 Tips regarding our infrastructure usage

Uses **SharePoint and MSTeams**

> DEV Team Service map / Misc. Procedures for our internal INFRA

DEV Team General Project Management

- 🦸 Follow up and user communication on every ongoing projects
- One chapter for each projects
- Each project will contain:
  - The test results to measure the development quality
  - Important meetings notes to share among us
  - User documentations and FAQs
  - Meeting presentations
  - Gantts and other organizational items

Uses **SharePoint and MSTeams**

> How to conduct a sprint meeting: Sprint Meeting Checklist

DEV Team Management

- 📊 Processes and team management knowledge
- 📈 More general project planification and gantt diagrams

Uses **SharePoint and MSTeams**

[Azure DevOps Board]

- Our Agile Board
- All of the planification of our work

> How to use it : Our Azure Board

[Code repositories novalixofficial]

- 🗂 All of our code : and the documentation addressed to the devs
- Each repository will contain :
  - README
  - Project Inputs/Outputs doc
  - Conception diagrams and notes (.excalidraw.png or .drawio.png)
  - List of testsheets sent to users (but not their results)
  - Example UI and mockups

For all heavy-duty general file storage, go to the NASSYN provided by the IT :

[IT DEV Channel]

Microsoft Teams discussions and files storage

Join with code : **b444l6f**

```
1 nassyn.novalix.local [192.168.90.200]
```

Then the path is :

```
1 //nassyn/IT/DEVS
```

Hardware Dependencies

PS : If you find a bit of documentation that’s misplaced, please store it in the appropriate folder, or ask somebody where it should go !

Our Azure Board

The Azure board is a complete and powerful tool to **enable us organize our work**💼.

But it can **only work if we feed it** 🍖 our everyday organisation items that pop continuously into our minds during our day to day development, user meetings and discussions.

See Agile principles

Pacing of the Azure Board Usage in the team's life :

👨‍💻 Everyday

- **I have a thought about a new feature that we should implement** -> New Feature with the tag "TBD", sorted under the correct Epic, under the correct Area
- **I need to create a new user story for some additional work that I'm going to do** : New User Story, ideally defined with the correct format, sorted under the correct Feature, linked with the other related USes, assigned to you, under your current sprint
- **I finished developing a task** : I look up the US #id and add it at the end of my commit with Git, I pass the tasks in status "Closed" in my sprint board ("Resolved" if you still need to test it in pre-prod)
- **I completed a User Story** : On the Team User Stories Board, I pass the US in status "deliverable" in my sprint board, and "closed" when the sprint ended
- **I have a comment to make about a task / I have a question about a US / a task** : Write a comment under the specific work item and @ the targeted person
- **I worked on something that popped up and wasn't planned** -> New Support ticket with a quick description on what's to do

👨‍💻 + 👨‍🔬 Every user interaction that creates a request / demand

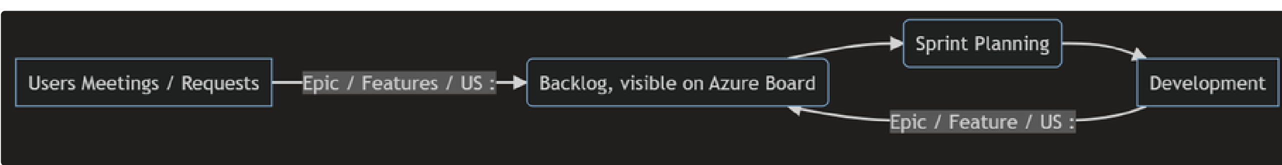
- Each meeting or discussion with the users must create new **documented Features** or new **redacted User Stories**. You can redact them in the Team Backlog
- The created user stories may have the "TBD" tag, and must always be in **Approved** state, meaning that the user have approved the request but the devs haven't yet validated it.
- This will allow to easily sort the different User Stories in the Team User Stories Board or on the Sprint Taskboard

📆 Every two weeks : Day of the Sprint meeting (Friday)

- I create the missing Feature / US or complete the Us that dont have any descriptions that I plan on presenting.
- The Product Owners (often us, the devs) meet with the key users. We go through the Feature / User Story backlog together and organize the next sprint. We can, in a second time, discuss about the second next sprint.
- ❌ We try to not create any work item in the Azure Board during this meeting ! Plan another meeting if you note that a lot of "non-trivial" tasks are being added.
- After that we evaluate, between devs with the *poker planning method*, the complexity of the tasks and write it on the US
- We makes sure to conduct our meeting according the ☑ Sprint meeting Checklist

---

Feeding / Consuming process of Azure boards :



User Stories Effort Evaluation

Effort evaluation

You have the liberty to plan your efforts as you see fit. But during the Sprint Planification, we will evaluate the User Stories as **Story Points**.

They are arbitrary units that we use to describe the **perceived** complexity of a UserStory.

By using them repeatedly, we get a sense of which **amount of complexity** one person or one team, can accomplish during a sprint.

Here is the Link to the Story Point Achievement Tracking Board (broken). Be sure to normalize the amount of SP to 5 days when you report the Story Point Achievement

Here is the link to the Scrum Metrics excel .

Internal Agile Implementation

An Agile project is organized in 4 layer of work items :

Level 1/2 - Organisation

1 - Epic

- General objective, can be a set of features of specific tasks tied together by a theme.
  The epic is limited in time (it's a rough estimation of the end of the Epic, more like an objective).

Examples :

> Jan. 2021 Release / VDR needed features / Full Bar code process

1 - Quest

- General objective that is not limited in time because it's cycling over and over. It's a way to track regular actions that we should perform in the team

Examples :

> DEL daily operations

2 - Feature

- Specific objectives linked from a technical and user-usage viewpoint

Examples :

> Inventory Management / Molecule export / Full Admin BackEnd

---

Level 3 - Description and Evaluation

3 - User Story

- Informal, natural language description of features of a software system. It explains a user focused feature in the most exhaustive way possible so that it's easily implementable / testable / validable. It is written in a specific format :

```
As a [persona]:
Who are we building this for? The chemist, the manager, the general user ?

I want to:
Describe the action that the user wants to do **from its point of view**,
not in a developer--that-will-implement-it mindset
This is the feature

So that:
how does their immediate desire to do something this fit into their bigger picture ?
What’s the overall benefit they’re trying to achieve ? Who need this ?
What is the big problem that needs solving ?
The feature answer a need, what was that need ?
-----
Acceptance criteria :
The user story is done when the user can do its action and, with the combination of all the right user stories,
Should the data appear in a list in front end ? Other data should be modified and visible ?
Should some side effect be executed (like a history for example) ?
Should an error appear if bad data are sent ? Which format should have the error ? in which case ?

```

Example :

> **As a** Simple User
> **I want to** login on enovalys with my novalix IDs
> **So that** I can use the labbook for my research reporting

> **Acceptance Criteria** : I can login with my novalix IDs in eNovalys && I can't login into enovalys if I don't have my IDs

3 - Support Ticket / Support Item

- Item that popped up during the project that required attention and consumed StoryPoints. It may be part of a project or just some general support.
  The goal is to track how much of our work was planned and how much was spontaneously added afterwards.
- You can assign a value to them right after finishing it

Examples :

> Export data for .. // Resolve production bug // Helped team member for <Project> ..

3 - Bug

- Like a US but can occur during a project. Can be planned or spontaneous.
- If planned need to assign a value to it at a sprint meeting, otherwise just quantify it on the fly

Examples :

> Add Field to interface / setup dev Env / create DB / Order Product / etc...

Level 4 - Personal work breakdown

4 - Task

An implementation Task, there can be as many a wanted, linked together as desired. They are here to fulfill one or several user stories [ref needed]. They can go through several state like : Conceptualisation / Developing / Testing / Deploying / Discussing etc...

Examples :

> Add Field to interface / setup dev Env / create DB / Order Product / etc...

Ressources :

Epic examples

User Stories Example

Difference between each 4 items

https://www.visual-paradigm.com/scrum/theme-epic-user-story-task/

Sprint Meeting Checklist

Here is the Link to the Story Point Achievement Tracking Board

Sprint Meeting Checklist

The day before, Alone

▢ 🎞️ The demo is prepared and you have already done this demo the day before
▢ 🎞️📃 The test sheet is redacted
▢ 🔄📜 You should have your **Sprint Backlog** ready with what's been done or not.

Pre-requisite

▢ 👦👩 You, the Product Owner, The DEV Team
▢ 📜 The **Backlog** of User Stories is complete and detailed, with all the work coming from various meetings with the users.

With PO and Users - Sprint meeting & Demo

▢ Display your progress with the help of your 🔄📜 **Sprint Backlog**

- Were there roadblocks
- What new topics were included in the sprint that wasn't planned at the start

❌ Don't mention Story Points

Don't mention the (dev) tasks inside the user stories

Don't justify any lateness of tasks that isn't caused by the users

▢ 🖍️ Note the reason for outperformed or underperformed sprint. Speak only with the users about what concerns them **directly** (needed information from them, etc...)

▢ 📦 Demo of the finished work, so for **EACH** US :

- Take one US
- Explain the acceptance criteria
- Display how it has been implemented
- Collect feedback in the form of new user stories or **plan a meeting** to collect feedback in-depth
- Repeat

▢ 🔄 Plan the next sprint together with the use of Feedback AND 📜Backlog

- The closed US if you forgot to close some
- The US that will be moved back to the backlog due to lack of urgency

⚠️ No TBW or 'New' US can be present in a set sprint. Ideally, no TBD either.

- Move unfinished but to continue USes to next sprint
- Add new one **from backlog** to complete the sprint

▢ Ask for priority in the displayed tasks.

Alone - US Brush-up / Self-improvement

▢ Store the created US under the correct Area Path / Epic

▢ 📑Correct any missing information or incomplete User Stories (TBW)

- ⛳ Explain / precise their goals so that the whole team can understand what te are story pointing
- You may want to start to create Tasks to better subdivide your US

> ⚠️ Don't add new rules that haven't been validated by the users! Only explicit what you've just discussed or the technicalities of the US

With the DEV Team - Story Pointing

▢ 🏷️ Evaluate the User Stories with Story Points

▢ 📏 Tailor your sprint as you think you can realistically achieve it.

▢ 🎭 We validate together that we all followed this checklist principles or what we need to change

▢ 📨 Redact summary email with definitely planned tasks to the PO / users

▢ (mandatory) We give each other a pat on the back for being such pros 🤝🙌

> 💪 Cheer up guys !

General Glossary and Acronyms

We are developers, working among chemists, biologist and other exotic jobs.

We'll need to be sure of the term we use and hear !

Area Path

Dashboard

[Azure]
Add widgets and queries onto one single dashboard.
This allows you to simplify your everyday Azure experience

DEL

=> DNA Encoded Library
Technology that used DNA to generate millions of chemical libraries and subsequent data.
It's also the name of the novalix pole whose main task it is

ELN

=> Electronic Notebook
The tool the chemist use to report their experiences and production. It is at the center of their work.
We have our internal ELN : eNovalys, but several others are used in Novalix.

Iteration

[Azure] ... We don't really know at this point 😅

SDF format

SDF (Structure Data File) where structure mean chemical compound; is described here : W Chemical table file (read both “mol” section and “SDF” section)

Important Notes:

- When we create / store SDF file (or mol files) we want to use the **V3000 format** (do not use the V2000, it does not correctly code the Stereochemistry)
- the “mol” format described in the documentation is often referred to as “MolBlock” in our code, especially when using RDKit.

SMILES

Smiles stand for **S**implified **M**olecular **I**nput **L**ine **E**ntry **S**ystem.

We will prefer to use the extended SMILES from chemaxon, when needed (stereochemistry and / or)

A smiles files (.smi or .smiles) has the following format:

```
Smiles	Name	property1 property2
CCl	Methyl chloride	12.5	SNF
CO	Methanol 5 NVX
```

The separator of the filed is a tabulation.

Tags

[Azure]

Additional informations on a work item for easy state description.
Currently we have **TBD** (To Be Defined) and **TBW** (To Be Written)

[DEL]

A chunk of DNA added to a library to identify it, each library is identified by its tags combination (A+B+C)

Query

[Azure]

Very complete tool to create views of all works items inside Azure, cross-project or not.

Resources

What's a project

Quickstarts

Refer to this article to know which computer configuration you should have to be able to work appropriatly at NovAliX : Hardware/OS specifics info to work at NovAliX

Refresher Course for new members

Order of exercices to follow to be u-to-date to our current dev practices. The goal it to detect any question you might have on our development practices.

It can be either Very specific on a how to write some piece to code, or very generic on the mechanics in play during a Front/Back Rest communication.

The Lead Dev is here to help as well as any other team member !

---

1 - Documentation light-reading

Read all of our Team knowledge article. Yes its boring but try to gather at least main key point of each articles.

Here is the recommended order :

- Our Internal Agile implementation
- Our Taskboard
- Global Knowledge
- Dev to ops process
- Versioning your projects
- Code re-usability

---

2 - Install necessary tools

- Developer QuickStart

---

3 - Technological Backend Training

We decided to follow somes popular technologies to ease our new members into the team and exploit the already existing librairies existing :

- ReactJS / Redux / MaterialUI
- Python / Django Rest Framework
- Docker

Here is a roadmap to go through all the topics used during your development :

3.1 - Python

- Create a python project in which there are 4 files :
  main.py / version.py / Worker.py / Desktop.py

version.py comes from the Common Team repository

Worker are :
Job, Name , Salary
string function to pretty print
use a dataclass to make the class

Desktop are :

- Nb Place
- Workers inside
- one method to add a worker
  - If No more place, a message must be displayed and no worker added
- One method to list worker presents

main.py must :

- Create 2 desktop
- Create 5 Worker
- Try to add a worker to a desktop with no places left
- Each time you run the program, the current version is updated the a message displays it

3.2 - Python Flask : A server

- Use Conda to create an environment and use it to install flask.
- Use the config.py file from the Common Team repository to prepare a JSON environment file
- Start the default app with version.py and the config.py. The environnement must be decided by a environnement variable from your system or your IDE.
  If the variable says "DEV" a message prints "I'm in DEV ENV" if the environnement says "PROD" the message says "I'm in PROD ENV"
- Everytime you run the project, the version.py is updated
- Write a server that exposes the following API :

```
1  openapi: 3.0.0
2  info:
3    version: 1.0.0
4    title: Sample API
5
6  paths:
7    /:
8      get:
9        summary: Get Hello world message
10       responses:
11         '200':
12           description: Hello world message with server version and config
13           content:
14             application/text:
15               example : "Hello world, I'm at version 0.0.1"
16
17   /files/:
18     post:
19       summary: Create a file with the provided name and content
20       requestBody:
21         required: true
22         content:
23           application/json:
24             schema:
25               type: object
26               properties:
27                 name:
28                   type: string
29                 content:
30                   type: string
31       responses:
32         '200':
```

```
33          description: File created successfully
34        '400':
35          description: Invalid request payload (content length exceeds 100), name already exists
36        '500':
37          description: Something went bad on the serverside, join the execption
38    get:
39      summary: Get file list by criteria
40      parameters:
41        - name: name
42          description: Will match any file where the name of the file contains the param value
43          in: query
44          schema:
45            type: string
46        - name: content
47          description: Will match any file where the content of the file contains the param value
48          in: query
49          schema:
50            type: string
51      responses:
52        '200':
53          description: List of file names that match the criterias
54          content:
55            application/json:
56              schema:
57                type: array
58                items:
59                  type: string
60                  description: File name
61        '400':
62          description: |
63            parameter doesn't exist (if you put anything else than name  or content as query param:
64
65  /files/{name}:
66    get:
67      summary: Get file content by name
68      parameters:
69        - name: name
70          required: true
71          in: path
72          schema:
73            type: string
74      responses:
75        '200':
76          description: File content retrieved successfully
77          content:
78            application/json:
79              schema:
80                $ref: '#/components/schemas/FileResponse'
81        '404':
82          description: file not found
83components:
84  schemas:
85    FileResponse:
86          type: object
87          properties:
88            name:
89              type: string
```

```
90            content:
91              type: string
```

PS : This format is called `.openapi` and is used to describe APIs in a very explicit way, you can use the swagger editor with this api to enhance your developer experience when communicating informations about APIs.

3.3 - Django Rest Framework

Do the same Exercice but with the framework Django-rest-framework

- You won't write file on the system anymore but create a new entry in the database, in a table of your choice
- Use the SQLite driver for managing the DB
- The api must work in the same way as 3.2

3.4 - The NovAlix boilerplate

- Start boilerplate BACK projet
- Add db + config
- Redo the same exercice (you can copy / paste code) with the boilerplate
- Add a route that exposes :

```
1  /files/all/:
2      get:
3        summary: Get all files
4        responses:
5          '200':
6            description: File content retrieved successfully
7            content:
8              application/json:
9                schema:
10                 type: array
11                 items:
12                   $ref: '#/components/schemas/FileResponse'
```

---

4 - Technological Frontend Training

We will use the end of the Python exercice to feed our app.

4.1 - React

Use React to create :

- `localhost:3000/` home page listing all available file in the back

When you click on one file, it will take you a detail page, :

- `localhost:3000/file/<file_name>` fetch the content of the file, and display it to the user

You also have a link in a header bar to create a new file :

- `localhost:3000/file/new` a form to input name / content + Submit Button + Cancel Button

4.2 - React + Redux

- Install redux to your project

- Do the exercice but now, except for the creation of a file, all data is stored in a redux slice.
- The slice must be named 'fileSlice'
- The slice must have at least two property :
  - 'displayed' for displaying the current file being displayed
  - 'allFiles' for storing all of the files fetched and displaying them
- +Bonus point for making a filter in the front to filter through the list of files

---

5 - Boilerplate

- Use your boilerplate with the Backend written from the previous exercices
- Redo the 4.2 exercice with the Front react boilerplate.

---

6 - Conclusion

Congratulation !! You achieved using the full boilerplate and are now ready to start on a serious project. Don't worry, your colleague can help you for any problem you may still have with the BP. They suffered through by the past 😋

Welcome to the team !

Python Environments

Specify the environment you want to work in with :

To be sure that you have a well setup environnement use the command :

`conda list`
It will display the list of packages presents in your env

If you want to freeze in a file such as **requirement.txt** use can use the reliable package **pipreqs** with `pip install pipreqs` and execute `pipreqs <PATH_TO_YOUR_PROJECT>`

`<PATH_TO_YOUR_PYTHON_EXEC> -m <APP_TO_START>`

Example :
The command `C:\ProgramData\Miniconda3\envs\crawler_env\python.exe -m pip install -r .\requirement.txt` will install all the requirements in the `crawler_env` environment , independently of if this environment was created with venv / conda or others.

Add new environments:

Environment creation

`conda create --name myenv`

Activate the environment

`conda activate myenv`
Then you can make install in youre env

Local Docker database

Postgres

Postgres Docker container 🐳

We will be using datagrip to generate the dump file and then use it to create our copy in a docker container

---

1️⃣ pgdump 🐘

Once we are connected to our Postgres database in DataGrip(new>data source>PostgreSQL), we can generate a dump file

- Right-click on the database(so on the second level, just below the data source)
- Export with pg_dump

🚩 The "Path to pg_dump" should point toward the pg_dump.exe in your PostgreSQL install(usually in Program Files).

2️⃣ Creating the docker container

The first thing to do is to pull the Postgres Docker image.

```
1  docker pull postgres
2
```

Then we can create our Docker container and name it Postgres.
We are mapping the container port with the host port so that we can access the database from the host operating system.

```
1  docker run --name postgres -e POSTGRES_HOST_AUTH_METHOD=trust -p 5432:5432 -d postgres
2
```

On success, the cmd will output the container ID.
The container should be visible and running in Docker.

3️⃣ psql

Now we shall connect to our docker container in data grip on localhost:5432 (new>data source>PostgreSQL)
We can then restore the database with our dump.

- Right-click on the container datasource
- Restore with psql

🚩 Same as with pg_dump, the "Path to psql" should point toward the pg_dump.exe in your PostgreSQL install.

This should™ recreate the Postgres Database inside your Docker container and it should subsequently appear in DataGrip

Mongodump

Mongo Docker container 🐳

We will be using datagrip to generate the dump file and then use it to create our copy in a docker container

---

1️⃣ Mongodump 🍀

Creating a dump from a MongoDB database.

```
mongodump -- host 192.168.xxx.xxx:yyyyy --username admin
```

This will prompt you to enter the password of the user if needed.
🚩 This usually takes a while, but the output is very verbose.

On success, it will create a dump in the cmd folder containing the dump files.

2️⃣ Creating the docker container

The first thing to do is to pull the MongoDB Docker image.

```
docker pull mongo
```

Then we can create our Docker container and name it mongodb.
We are mapping the container port with the host port so that we can access the database from the host operating system.

```
docker run -d -p 27017:27017 --name mongodb mongo
```

On success, the cmd will output the container ID.
The container should be visible and running in Docker.

We will now copy the mongodump into the docker container.

```
 docker cp dumpPath  containerName:/dump
```

🚩 This might take a while and does not output anything while it is working.

3️⃣ Mongorestore

Then we will be able to launch the restore from the newly created dump folder in the Docker container.
First, we launch a bash inside the container

```
docker exec -it containerName bash
```

Then we may launch the restore

```
mongorestore
```

This should once again take a while.

Once this is done you should™ be able to access the database from localhost:27017

How to install a flask service on IIS

This tuto is a simplification of the tutorial : Deploying Python web app (Flask) in Windows Server (IIS) using FastCGI

1 - Installing CGI on IIS 7 / 10

Go to Server Manager then:

- "Add Roles and Features" in Server Management
- Do a Role-based or feature-based installation
- In the "Server Roles" part, check Web Server (IIS) -> Web Server -> Application Developpement -> CGI
- "Next" until "Install"
- once done close the window

2 - Install WFastCGI

- Open a terminal in admin mod and enter the following command to install wfastcgi and activate it:

```
pip install wfastcgi
wfastcgi-enable
```

- Copy / paste the `wfastcgi.py` that has been added (normaly it's added in the following folder: `C:\Program Files (x86)\Python37-32\Lib\site-packages`)
  paste it in the folder `C:\WebSites\[your_service_name]`

2 - Install a new site in IIS to store the flask application

2.1 - Install the site

- open IIS
- add a new site with the name **enovalysFlaskServer** (check that the application pool is created with the same name)
- as the physical path put `C:\WebSites\[your_service_name]`
- select https and put 6767 as port the host name is `www.soft-enovalys.com`
- be sur to have the port 6767 open in the firewall

2.2 - Open the port to the azure firewall

- connect on azure portal: Microsoft Azure
- select the virtual machine **enovalysapp**
- go to **point de terminaison**
- add a rule for flask where public and private port are 6767 (with TCP)
- validate

2.3 - Add an handler

- select this new website and click on **Handler Mappings**
- click on the right and **Add Module Mapping**
  - as request path put `*`
  - as module part select `FastCgiModule`
  - as Executable (specify youre env if possible), select : `"C:\[PATH_of_youre_python_env]\python.exe"|C:\WebSites\[your_service_name]\wfastcgi.py`

  - as name: `PythonFlaskHandler`
  - click on **Request Restriction** and check that "Invoke handle only ..." is **uncheck**
  - click on **"OK"**
  - on the window open click **"Yes"**

2.4 - Configure FastCGI

- go to the root server settings and click **"FastCGI Settings"** you may have to quit and open again IIS manager
- select the appropriate CGI configuration (double click on it)
- in the opened window, click on **Environment Variable** -> on the 3 dots next `(Collection)`
  - add the 2 following variables:
  - `PYTHONPATH` with value : `C:\WebSites\[your_service_name]`
  - `WSGI_HANDLER` with value: `app.APP` as our application is called **app.py** and the main Flask application is named **APP**, be aware that the cass is important
  - validate twice (button OK)
- open services and stop the python crawler service if it's running
- launch the site and click on the link

Troubleshoot

Env Var

Makes sure you have the correct Python and python Scripts path added in your global PATH

WHEN INSTALLING python version :

Check that python is installed for all users, otherwise FastCGI and IIS won't be able to acces the python exec at the right place,
Install everything globally, including module pip, and etc...
And restart the machine

If Cors error with flask :

The Cors(APP) instruction should be enough but if you have the rror stating :

> Access-Control-Allow-Credentials' header in the response is '' which must be true'

Then add the header 'Access-Control-Allow-Credentials' set to 'true' in IIS in the Site "enovalysFlaskServer", in the "HTTP Headers" section

VSCode tips

- Usefull plugins :
  - Django
  - Python
  - Docker
  - Live Share
  - Markdown All in One
  - Postman
  - YAML
  - Github Actions
  - Python Environment Manager

Usefull settings :

- Search for ‘Wrap Tabs’ in the settings (Ctrl+Shift+P) and activate it. This will stack open files in the editor instead of making you scroll horizontaly.

Use your conda env :

Ctrl + Shift + P → opens a parameters research dialog
Look for ‘Python : Select environment’, you should see your conda envs, select the one you want to use
Ps : If your app uses env variables, you can add them to the conda env so you don’t have to type them in each time you open a terminal. Commands here : Managing environments — conda 23.9.1.dev39 documentation

Flow for VSCode :

Flow is a JS static type checker

https://marketplace.visualstudio.com/items?itemName=flowtype.flow-for-vscode

Make sure flow-bin is installed in npm packages

- Install the Flow extension
- On VSCode : Ctrl+P and look for the file '.config/Code/User/settings.json (just type settings and you should see it)
- Add a line : '`"javascript.validate.enable": false`'
- Add ‘.flowconfig’ file to the root directory
- Reload VSCode if nothing moves

Your files should now be staticaly checked by Flow

Git tips

Branch cleaning on your computer

- Fetching all remote branch
- Clearing the one that dosen’t exist on remote
- Removing all the local branch that doesn’t track a remote branch

```
git checkout master

git fetch --all --prune

# list the branch that you can delete on your computer because already merged somewhere
git branch --merged

# delete them one by one
git branch -d <copy/paste from the previous list separated by a space>

# then git branch --all to see if there is any leftover branch
#
# it's possible that a branch that had its origin pruned is still present an
# and doesn't appear in `git branch --merged` because it's unmerged
#
# IN THIS CASE first make sure its defently merged somewhere, or that its LAST COMMIT
# is present on another remote branch
# THEN
git branch -D <name-of-branch-not-merged-but-already-has-another-remote-on-the-same-commit>
```

Git Tag management

- Creating a tag

```
git tag v1.0.0         # Creates a lightweight tag at the current HEAD
git tag v1.1.0 1a2b3c4  # Creates a lightweight tag at commit 1a2b3c4
git tag -a v1.2.0 -m "Release 1.2.0" 2b3c4d5  # Creates an annotated tag with a message
```

- pushing a tag

```
git push origin v1.0.0  # Pushes the tag named v1.0.0 to the remote 'origin'
```

- removing a tag

```
git tag -d v1.0.0  # Deletes the local tag v1.0.0
```

- push the removal of a tag

```
git push origin --delete v1.0.0  # Removes the tag v1.0.0 from the remote 'origin'
```

Software Development Practices

Why

We want to develop together, even when we're not developing on the same project.

To achieve that code coherence across our project, it's necessary that we **discuss and nurture our common software development knowledge together** at all times.

To do that, we highly encourage you to **write and discuss with the team** about all practices that you deem important for all the team to be aware about and apply.

Please do so here in this wiki chapter and don't hesitate to openly comment or edit the already written documents to make them better for everyone.

Which Dev Flow to use

💻 Versioning with Git

We use GitHub For all of our Projects.

We have target deployment machine on the cloud as well as on-premises.

We use as much the **Git CLI** as softwares like **git-extension** or the basic **gitk**. Use whatever you want as long as you are still effective.

🔂 Use Conventional Commits

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
[Tags for Azure: AB#<azure-task-number>]
[Tags for Jira : <JIRA-ISSUE-KEY> #comment to add a comment]
```

- `type` can be one of `[feat | fix | build | chore | ci | docs | style | refactor | perf | test | ...]`.
- `scope` is optional and can provide additional contextual information.
- `description` is a small description.
- The `body` should explain what you wrote and why. If the solution was complex and not trivial, explain the principles used to resolve the problem to help the next person reviewing your code/commit.
- `footer` can be used to reference issues, mark breaking changes, etc.

Examples:

Short one :

```
feat(lang): add Polish language
```

Detailed one :

```
fix(rest-usage): prevent racing of requests

Introduce a request id and a reference to the latest request. Dismiss
incoming responses other than from the latest request.

Reviewed-by: Z
Refs: #123 EN-1076 #resolve
```

Note that in Conventional Commits, a commit of the type `fix` patches a bug in your codebase, a commit of the type `feat` introduces a new feature, and a `BREAKING CHANGE` introduces a breaking API change. This convention correlates with Semantic Versioning.

ℹ️ Why Use Conventional Commits

- Automatically generating CHANGELOGs.
- Automatically determining a semantic version bump (based on the types of commits landed).
- Communicating the nature of changes to teammates, the public, and other stakeholders.
- Triggering build and publish processes.
- Making it easier for people to contribute to your projects, by allowing them to explore a more structured commit history.

💡 **Glossary of commit types**

- `feat` Commits, that adds a new feature
- `fix` Commits, that fixes a bug
- `refactor` Commits, that rewrite/restructure your code, however does not change any behaviour
  - `perf` Commits are special `refactor` commits, that improve performance
- `style` Commits, that do not affect the meaning (white-space, formatting, missing semi-colons, etc)
- `test` Commits, that add missing tests or correcting existing tests
- `docs` Commits, that affect documentation only
- `build` Commits, that affect build components like build tool, ci pipeline, dependencies, project version, ...
- `ops` Commits, that affect operational components like infrastructure, deployment, backup, recovery, ...
- `chore` Miscellaneous commits e.g. modifying `.gitignore`

❓ When to Branches, Commit, Pull Request, etc…

Given the size of your project, you may use one of the two main used versioning flow :

- Git Flow for “complex” projects
- GitHub Flow for “simple to medium” projects

♻️🚚 Link to CI/CD

You developments will directly feed the CI/CD process explained here : CI/CD Process

---

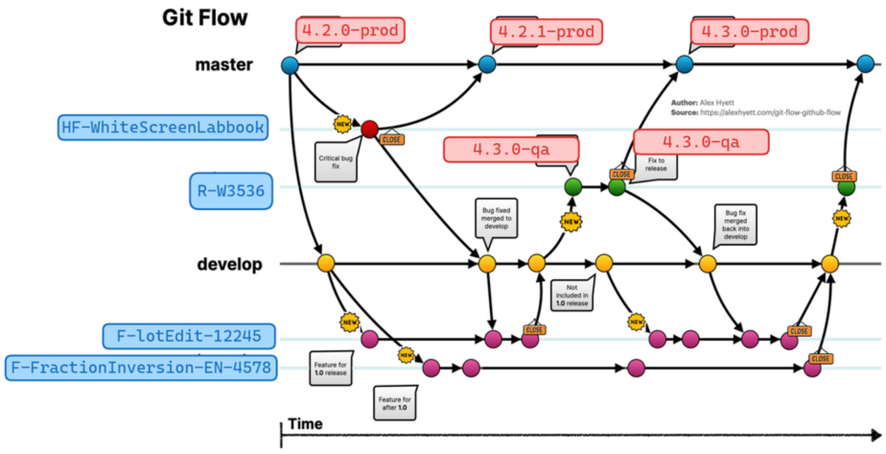
Source CI/CD : Continuous Integration and Continuous Delivery (CI/CD) Fundamentals
Github Actions : Understanding GitHub Actions - GitHub Docs

Git Flow

ℹ️ When you're project have **complex multiple releases** that have various **dependencies** (time, users, hardware, …) that dictates their releases.

The branches give the visibility on which **release** contains which **features**.

The tags give the visibility on which **code** is running **where.**



🔥 Hot fixes

Quick bug fix following a deployment on production.

- Create a new branch `HF-<nameOfFix>`
- no need to be tested by user
- **Merge** to prod while **Updating** `IHM/locales/release.json` with the new patch number `X.X.+1`
- **Merge** to develop
- **Deploy** before creating the 🔖tag on git
- **Increase the 🔖tag patch number** `X.X.+1-prod`
- **Delete** the `HF-<nameOfFix>` branch

🏃 Start new sprint

- Create new 📦 **release branch** `R-<nameOfRelease>`

👨‍💻Start new dev for a clear objective (📦 release, 🏃 sprint)

- Create **one** new branch `F-<nameOfFeature>-<(azure[XXXX]|jira[EN-XXX])>` for **one** feature, US, bugfix, refac, etc… from **develop**
- **Write a new bloc** in the `IHM/locales/release.json` with the bug/US you are developing (understandable for the basic user). The bloc should **not have a version an date**
- **Work** hard, play hard until ready to **put in a 📦 release**
- **Merge** current branch into **develop** + **Merge** into 📦 **release branch** `R-<nameOfRelease>` (merge your `release.json`)
- **Delete** the `F-<nameOfFeature>-<(azure[XXXX]|jira[EN-XXX])>` branch

🧪 Put a release branch in QA for users

- **Select the release to put into QA** `R-<nameOfRelease>`
- **Update** `IHM/locales/release.json` with the version `X.+1.0`
- **Deploy on QA** before creating the 🔖tag on git
- 🔖tag the appropriate version `X.+1.0-qa`

🎉 Put a Release to Production

- **Merge** 📦 **release branch** `R-<nameOfRelease>` **to master**
- **Report** the 🔖tag of the release `X.X.X-qa` to production, changing the end to `X.X.X-prod`
- **Deploy on Production** before creating the 🔖tag on git
- **Delete** the release branch `R-<nameOfRelease>`

🧪👨‍💻 Edit Release from QA returns

- **Develop on the 📦 release branch** `R-<nameOfRelease>`
- **Deploy on QA** before moving the 🔖tag on git
- **Delete** the current 🔖tag `X.X.X-qa`
- **Create** the 🔖Tag `X.X.X-qa` to the **new HEAD of the branch**

- Source : Git Flow vs GitHub Flow , adapted for our usage with the tags
- Git commands for those actions : Git tips

🏆 Making your first impact to the project :

Introduction :

Before typing your first line of code, make sure you've fetched all the remote things (branches, tags…)

Pushing your branch :

After you commited everything and you finished your work on this feature, fix, etc… you will have to **push** your **local branch** to the **remote repository**.

Make sure you are on the right branch using `git branch` or `git status` .

Then push the branch.

- You can use **Git Extensions** or another tool to help you to manage your Git repositories or simply **use the command line** if you are more comfortable with it.
- Remember : **tools used at NovAliX**.

```
1 git fetch <remote> // Fetch all of the branches from the repository. This also
2 git fetch <remote> <branch> // Only fetch the branch.
```

<git fetch> - documentation.

After fetching everything and according to the ***Git Flow*** we're using, now you will have to **switch** on the `develop` **branch**, and after this you will **create** your **personal branch**.

```
1 git checkout develop // Switch to the develop branch.
```

You can make sure you are on the right branch, and list all the branches available using `git branch`.

Now you can create your new branch and work on it.

Remember to respect the naming conventions above ⬆️ !

```
1 git checkout -b <your_branch_name> // Create and switch to your new branch.
```

You can now start coding ! 💻🤓

Committing changes :

You will have to commit your changes at each steps of the development process. It's preferable to commit very often.

Recommended : you should commit whenever you want, when something is working, who doesn't break anything, fixed something…

Not recommended (don't do this) : commit when you finished all your work, it will create a commit for all of your different files you've created. Again don't do this.

It's recommended to `git commit` and `git push` (we will see this one after) to avoid losing your work.

We don't know what could happen 😉

In case of fire

1. git commit

2. git push

3. leave building

How to commit :

See what file you created, modified, all the things you need to commit.

```
1 git status // Show the working tree status.
```

Next you will add the file you want to commit :

```
1 git add <your_file_name> // Add file contents to the index.
2 git add . // Add everything, not recommended.
```

After adding your file, you can commit, using a message, describing what you did.

Remember to respect the naming conventions !

```
1 git commit -m <your_commit_message> // Record changes to the repository.
```

```
1 git push // Update remote refs along with associated objects.
```

You will probably be prompted to set the remote as upstream. Just do what is prompted.

```
1 git push --set-upstream origin <your_branch_name>
```

You now have pushed your branch to the remote repository !

- The **next step** is to **merge your branch** with the **release branch**.
- It's possible that the release branch is not created yet !
- If you planned to be absent, **YOU NEED TO COMMUNICATE** the fact that you have pushed your branch and ask for a merge in time !
- If the release branch is created you will have to merge your pushed branch into the release branch. This is the next step.

Merging your branch to the release :

You can fetch the remote branches to get the latest one, you will now have access to the release branch.

Remember to use `git branch` or `git status` to check your branch.

Now you can `git checkout <release_branch>`, you are now on the release branch.

Start the merge : `git merge -no-ff <name_of_your_branch_you've_just_pushed/worked_on>`.

There will be conflicts, **you need to resolve them** to complete the merge. Use the editor or the tools you want to resolve it.

After the conflicts are resolved you need to commit your changes, simply `git commit`.

You don't have the obligation to create a message here, one message is generated.

Push the changes : `git push`

You can now delete your branch locally and remotely :

`git branch -d <name_of_your_branch_pushed_recently>`

Delete remotely using Git Extensions or your tool.

You just merged your branch for the release !

Updating the release.json file :

You will now have to update the release.json file, you can do it before the merge if you want to.

Use the file as example and respect conventions, be clear on what you describe.

For my example :

```
1  // release.json
2
3  {
4      "LATEST_VERSION": "v4.8.0",
5      "DATE": "",
6      "ENHANCEMENT": [
7      ],
8      "FIXED_BUG": [
9        "List of Witness Experiment was slow to load."
10     ]
11 }
```

Still on the release branch, use what you learnt : `git status`, `git add`, `git commit`, `git push`.

If there are no problems you should have completed everything !

🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉**Congratulations ! You just made your first impact on the project !**🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉🎉

You can use AB# to link from GitHub to Azure Boards work items.

Usage : `git commit -m 'fix(witnessLoading): added release note after merging AB#4715'`

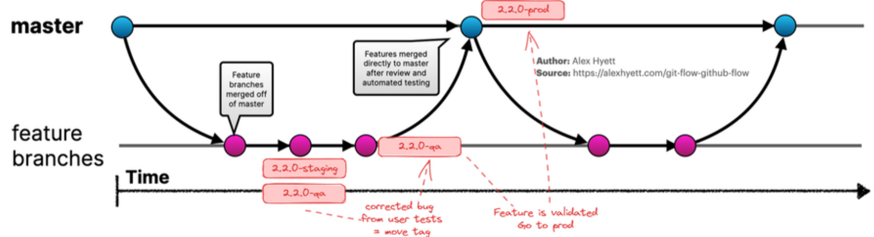
You will need to replace “4715” with the ID on the Azure Board.

Do this for every files, and remember to commit very often.

GitHub Flow

Work In Progress

GitHub Flow



🔥 Hot fixes

Quick bug fix following a deployment on production.

- Create a new branch `HF-<nameOfFix>`
- Check automated tests **before commit**
- no need to be tested by user
- Update your `version.py` with the version `X.X.+1`
- 🔖**tag patch number** on `HF-<nameOfFix>` with `X.X.+1-staging` to check that your commit won't break infrastructure
- Deploy manually if not done automatically
- **PR** to master
- 🔖**tag patch number** in **master** with `X.X.+1-prod`
- **Delete** the `HF-<nameOfFix>` branch
- Deploy manually if not done automatically

🏃‍♀️ Start new sprint

- Pick a **US** to start working on
- Create new 📦 **Feature branch** `F-<nameOfFeature-AZUREREF>`
- No need to prepare a Release branch, you will put your feature in testing continuously

👨‍💻Start new dev for a clear objective (📦 release, 🏃‍♀️ sprint)

- Create **one** new branch `F-<nameOfFeature>-<(azure[XXXX]|jira[EN-XXX])>` for **one** feature, US, bugfix, refac, etc… from **master**
- **Prepare** your `version.py` with version `0.0.0` because you don't know the final version beforehand
- **Work** hard, play hard until ready to **put in staging / QA**

🧪 Put a Feature in QA for user

- Update your `version.py` with version `X.+1.0` because its the target version for this staging build
- 🔖**tag with** `X.+1.0-staging` , if it already exists, just move the tag to your branch
- Deploy manually if not done automatically
- Check that nothing breaks
- → If 🔖 `X.+1.0-qa` don't exists, then🔖**tag with** `X.+1.0-qa`
- → If 🔖 `X.+1.0-qa` already exist, then PR your branch to the tag then **move the tag** to the **newly created commit**
- Deploy manually if not done automatically

🎉 Put a Feature in Production

- Add the tag `X.+1.0-prod` to the commit that already have passed the commits with the same version `qa` and `staging`
- Deploy manually if not done automatically

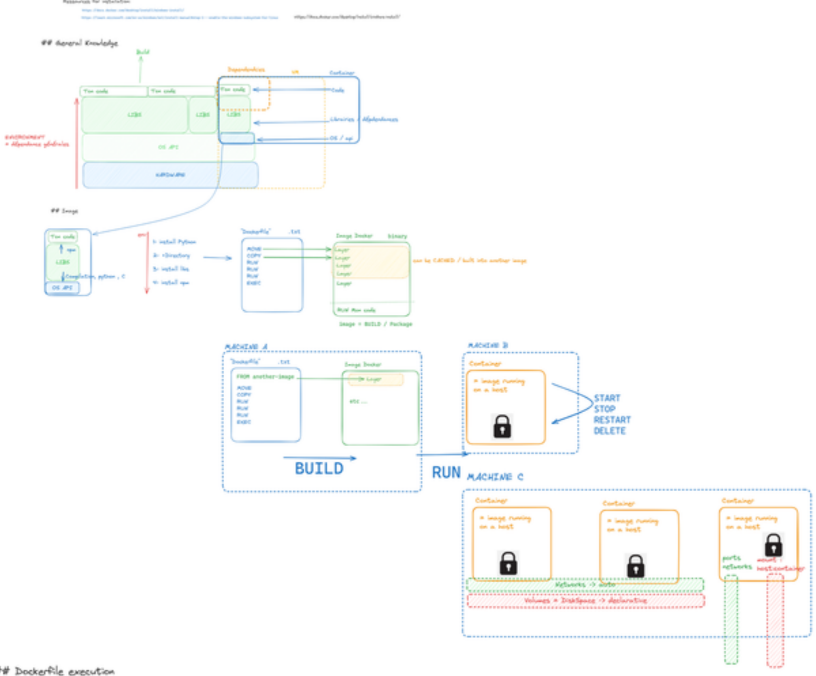
🧪👨‍💻 Edit Release from QA returns

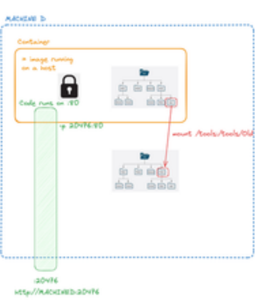
- **Change the code** on the branch that **contains the same** 🔖**tag** `X.X.X-qa` as the version your user are communicating to you about (make sure they can see the version so you don't get lost)
- move the 🔖**tag** `X.X.X-qa` to the new head of this branch
- Deploy manually if not done automatically

- Source : Git Flow vs GitHub Flow, adapted for our usage with the tags
- Git commands for those actions : Git tips

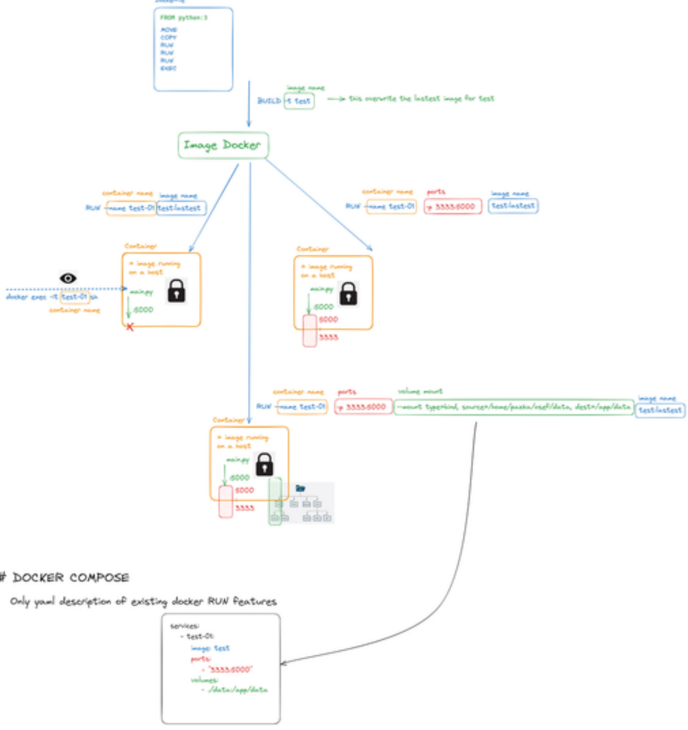
Docker Usage

Docker Bootcamp



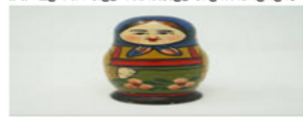


Dockerfile execution

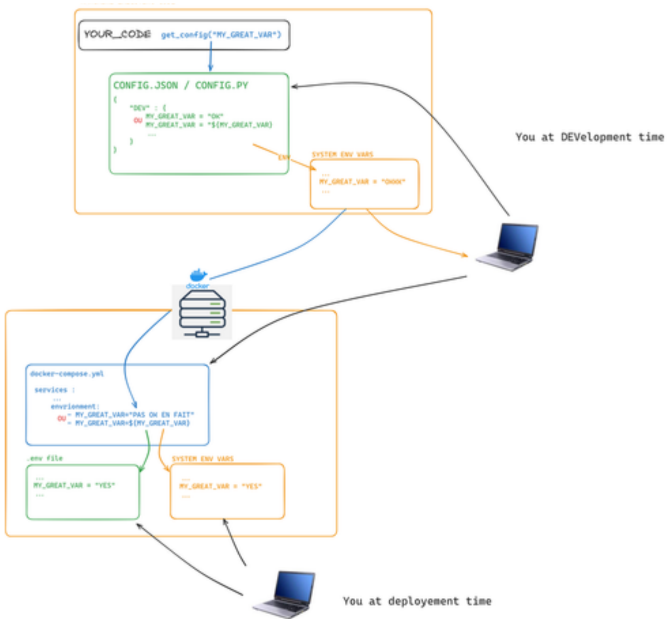


Env variables :

Le fun des variables d'envrionement :



MAHCHINE EXECUTING CODE



Exemple des commandes :

```
docker build -t XXXX .                                  // build le dockerfile (XXXX = nom de l'image) et le '.'
docker images                                           // récupère la liste des images docker
docker rmi XXXX                                         // supprime les images (XXXX = id)
docker ps -a                                            // liste des dockers qui tourne
docker run -d XXXXX                                     // lance le docker séparément
docker stop XXXXX                                       // stop le run du docker (XXXXX = Name / Id)
docker rm XXXX                                          // supp le docker (XXXXX = Name / Id)
docker run test:latest -p 3333:5000 XXXX                // Spécifie le port entre le pc et le docker
docker run -p 3333:5000 --name YYYYY XXXXX              // Donne un nom au run(process) du docker
Exemple de commande :
docker run -d -p 3333:5000 --name test-docker-01 test

docker run -d -p 3333:5000 --name test-docker-01 test --mount type=bind,source=/tmp,target=/usr
                                                        // Commande pour monter un espace (Volume)



//Curl GET et POST

curl -X GET http://localhost:3333 (sans le s au http)
curl -X POST -H "Content-Type: application/json" -d "{\"name\":\"test3.txt\", \"content\":\"test content\"}" htt

docker exec -it test-docker-01 sh                       //Pour avoir un bash dans le docker
docker image prune                                      // remove all unused image
```

Glossary

Container

A running `Docker Image` that can be altered by command line when 'entering' the container.
The alterations will not save the modified data between start/stop except for the data that are `mounted`.

Docker Hub

Public repository of standard and maintained images. The description contains the avalables `port` / `mount` / `env vars` and options for the building up of your container / dockerfile with this image

Dockerfile

Succession of specific docker `Docker Instructions` (see Docker file Cheatsheet) that will each create a layer to form a final Image.
A dockerfile often start from another public image from the `docker hub` (ex : postgre)

Image

Succession of layers, each of them being an docker operation that will add up to the final state of the container we want to run

Mount

Link between the inside of a container (src folder) and the host (dest folder).
The data will be read/written in the folder on the host instead of inside the container.
The apps inside the container have no knowledge of mounts and only see the folder inside the containers.

Dockerfile Cheatsheet

Dockerfile References

Architecturing projects

Planning the development of a new project in a stable and durable manner takes time, time which will be saved later in the process.

The time consumed will produce **discussions** and more importantly **documents** that will hold the very core documentation and stability of the software developed later on.

There are many ways of implementing this planning and the eNovalys team we want to it it that way.

⛳ Goal

By cementing our ideas and projects designs in **document formats**, we wan create a concrete, well documented foundation to hold our following discussions, changes of plan, features additions and other part of the project life.

A new project will likely use some of the original architecture's existing features, and need the creations of new ones. Thus it's important to keep track of the current states of different aspects of our designs, architectures, state of developments etc...

We follow this goal with the help of several concepts.
We don't follow any of them rigorously but borrow from them while keeping in mind the state of our ressources.

🔍 Concepts

The whole approach is based on several concepts :

- The MoSCoW Prioritization
- The 4+1 architectural view model
- Information System Urbanisation
- The Microservice Architecture
- Our Internal Agile implementation

Attention

All notes which arises from complementary discussions MUST be written in the associated items which brought up the discussion.

This aim to save a huge amount of discussion in this process which is design to create a maximum of discussion before the development of the project.

Tools

Draw.io : General purpose online Diagram Editor (a bit clunky)
XMind.net : Easy MindMapping tool

1 - End User

The End User knows best

All design processes must start with discussions with the end-users that will often describe their "Dream tool" without any regards for any software or physical realism.
And that's okay ! You job is to get from this discussions the following elements with as much clarity as possible.

In the current process/tool :

- What is satisfactory
- Why is it satisfactory (why it shouldn't change)
- What isn't satisfactory
- Why isn't it satisfactory

Then the question :

- **Which of those items truly need software development ?**
  Can they not be solved/fulfilled by just a different process design ? The increase or decrease of some ressources inside the process ? etc...

And finally :

- How would you classify those need by order of Must Should Could Won't

The goal is to bring the user to talk about the **real need to fulfill** and for you to reduce the amount of necessary work.

> Remember : The user knows the process better than you, you can trust him to give you the truth about the real need of the project, even when it doesn't seem important/logical to you.
> Only He, can help you create real value.

📰 Deliverable Examples
Notes and Comments

2 - User Stories

How to avoid re-describing the same thing over and over

(4+1 View => Scenarios)

All the user demands will result in a very concrete set of Use Cases written as User Stories. There can be a dozen or a hundred of US generated by this process.

Even if it's painful to write, this will become the source of truth for all following discussions with your end-users and will avoid a lot of headaches.

Try to include your notes in the user stories to explicit where they come from.

This set of US will also become the basis for the following steps of the design process.

> All comments and quirks of design which arises from complementary discussions MUST be written in the associated items which brought up the discussion.

📰 Deliverables examples
Azure DevOps User Stories

Once you have all of those user stories, you can start to group them in **Features** for easier ressource

2.1 - Create user rights table

Know who can do what

The confusing discussion that goes "Ha but I don't want [role] to be able to [feature], only this [another role]" will go on for as long as the project exists.

To prevent the pain linked to this problematic, solidify your knowledge of what are the existing roles, and which role have access to which features. This can easily be done with an excel and it will be great to discuss with users in the future and to update when you pick up the project later.

> Don't be afraid to write as much lines as possible in your document, the more exhaustive the better

📰 Deliverables examples
Excel joined in general doc folder (DevOps or git project) (broken)

3 - System Features

From user ideas to software

(4+1 = Logical View)

Once you have the set of necessary User Stories for the End User to fulfill its work, you can start to identify the **System Features** the system need to be able to support those use cases.

This should bring you to check the already written documents about the current existing software feature of the system.

Then you can start writing up the new features to create.

> All comments and quirks of design which arises from complementary discussions MUST be written in the associated items which brought up the discussion.

Not too quick

When you do so, you'll also start to conceptualize some aspect of the step 4 as you want to create big feature that hold a lot of functionalities and that's ok, you'll just need to keep those ideas not to far away. Just try to not hold on to it to much because it can cause some confusions in the process, in the mean time, explode this big feature.

📰 Deliverables examples
XMind of Software Features

4 - Software Components

Do we start with the Data Manager™ or the Service Controller™ ?

(4+1 = Physical View)

The more you specify you features, the more it should become apparent that some of them will use the same group of resources, same protocol of communication, same design patterns etc...

You group those in common **Software Component**

Those Software components will help you start with designing and developing one component at a time, thinking only about its constraint, deployment, contract with other components.

You guessed it, those components will become micro-services ! 🙌

Also, depending on your resources, you'll be able to start the development of one or more component at a time without having to think of the whole system the whole time. Bu that's for the next step.

> All comments and quirks of design which arises from complementary discussions MUST be written in the associated items which brought up the discussion.

📰 Deliverable examples
XMind of software component regrouping the previous features

Don't go too far, too quick

When grouping those components, you'll start to conceptualize the communication protocols and DB to use, code to write etc...

Like the previous step, don't let you focus to much on them as the goal is to design you component the way they make sense for the FEATURES and not the DEVELOPMENT implementation.

5 - The APIs

Should I POST you my DTO and you 204 me an OK or should WEBSOCKET me your ORM ?

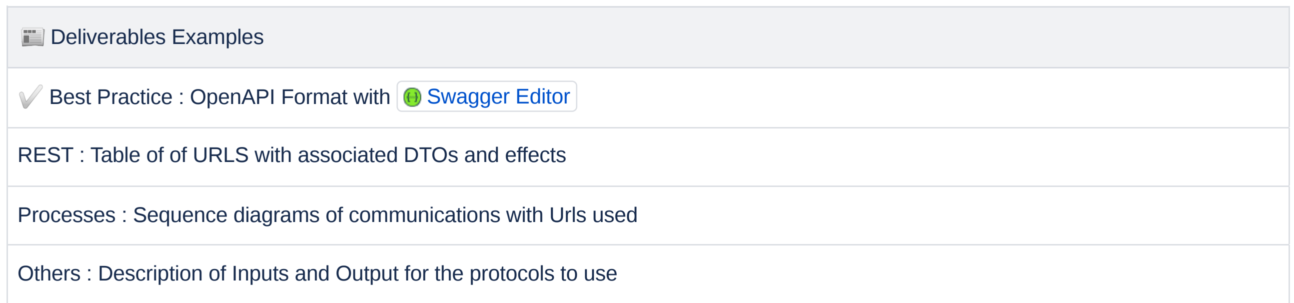
(4+1 = Process View)

The process view must be quite concrete after the previous steps and a lot of note should relate to this one.

The goal is to explicit the data flux between components for them to be able to fulfill all of the User Stories.

We are talking about **Contract of Communication** that we often read implicitly as APIs

> All comments and quirks of design which arises from complementary discussions MUST be written in the associated items which brought up the discussion.



📰 Deliverables Examples
✔️ Best Practice : OpenAPI Format with Swagger Editor
REST : Table of of URLS with associated DTOs and effects
Processes : Sequence diagrams of communications with Urls used
Others : Description of Inputs and Output for the protocols to use

Not too quick ?

We are very close to the concrete software development at this point and you may complete this design process at the same time as the Step 6.

Simply be aware that a **strong contract** will be used as a single source of truth between two microservices and as the user stories for end-users, the contracts will avoid headaches and redundant discussion with other developers.

6 - The Software

Praise the UML

(4+1 = Development View)

Now for the fun part, with all of the previous discussion and documents, you can start thinking about the implementation of your design.

You should end up knowing the necessary technologies and the Software Component that use them :

- DB - Tables
- Language - Classes - Tests - Design paterns
- Protocols (REST / Websocket)
- Parts of the Services Mesh used
- Hosting - Deployment

And maybe others I'm forgetting.

> All comments and quirks of design which arises from complementary discussions MUST be written in the associated items which brought up the discussion.

Deliverables example
DB entities : Entity Relationship Diagram (the easiest)
DB design, Code design : Class Diagram
fine-grained development documentation : Today I Learned for programmers
Sequence diagrams, MCD, and other doc can be put in a Resources folder, Boilerplate example (broken)

Hosting / Deployment and the Infrastructure

The "hardware" used should be mapped or at least reference with the available IT Infra documentation

Toolchain Setup

WIP tool chain

Start

Traduction setup

conf file copy

etc...

Auto build : example Cake, / gulp
https://dev.azure.com/novalix/Team-Knowledge/_git/code-utils?path=%2Fjs%2Fgulp-example.js&version=GBmain (broken)

Deployement : What to do

Develop to ship 📦

You should build an application with some principles in mind (Readability, Modularity, Logging, Error management, etc...) and to those you should add : *Deployability*.

That's to say that your app should be easily and quickly :

- Buildable
- Configurable
- Stoppable

You can achieve this by using :

- Single point of configuration (config.py / env.js)
- CLI parameter to specify last minute parameters
- Dependency specific : List every dependency required for running your app,
- Automatically buildable, with external tools like Cake(C#) or Gulp (Js), or even bash/batch if it works for you

Of course document the specifics of your configuration. This will make the app *conteneurisable* and easily manageable (start/stop) as a service.

Ship once, deliver everywhere 📬

Once you built your app, the deployment process should be done with an external tool dedicated to this task:

- Nginx
- IIS
- specific WSGI apps

See Quickstarts for detailed deployment processes

---

source : Why use nginx for flask or django

Versioning your projects

Every project should have some sort of versioning incorporated to the code so that you can check the version of the project at runtime.

0. Why

You need to be able to go on a website, check the version, then check your git repo to know, if this bugfix or this feature has already been implemented or not in the app your are watching.

To do that, the back should have a version, the front should have a version and you need to increment this version at each delivery !

For reference :

- **Major** - Rare ▶ Involve changes to the user interface or underlying architecture. Major updates often require thorough testing and may require users to learn new functionalities or adapt to changes in the software's behavior.
- **Minor** - Occasionally, end of sprint ▶ Smaller changes and improvements to a software product, such as bug fixes, performance optimizations, or minor feature additions. Minor updates are usually backward compatible with the previous version and do not require significant changes to user workflows or relearning of the software.
- **Patch** - Frequent, anytime ▶ Small, targeted update that addresses specific issues or vulnerabilities in a software product. Patches are usually provided as quick updates that can be applied without requiring significant changes to the software's functionality.
- **Build** - Each execution of the code ▶ Build is the number of time the project has been run/compiled. Used to determine how much work has received this update. Also can be used to trace how much time the software has been executed.

Here are the solutions selected for different language at Novalix :

1. All projects

- Have a `README.md` ! It will be the start of the documentation and is very easily accessible by every other developer. You should always have a `README.md` file at the root of your project

2. Javascript

Version number

Have a `version.json` file at the root of your project and use a `environment.js` module to store the json file as a variable

⛓ Version dependency

Use the `packages.json` provided by the package manager `npm`. If not present, create it : `npm init` and `npm install` everything you need.

Note : The `packages.json` file MUST be present in you git repository but NOT the `node_modules` folder.

3. Python

Version number

Use a small module to manage and manually or programatically update your version. That ay you will be able to easily manage automatic version increment when compiling you project. And in the mean time you can always change the version manually through this file.

Here is a module that you can use as is : version.py

⛓ Version dependency

First, make sure that you have a conda environment for you project
installation link: Miniconda3
Install for all user and add it to the PATH for use it in command line.

```
conda create --name <your-env> python=3.9
cd <project-path>
conda activate <your_env>
(once env is activated, and if file is present) pip install -r requirements.txt

```

PS : Always create a new environment for your project, if you're not familiar with conda, here is a cheatsheet

Then, make sure you have a `requirements.txt` file at the root of your project. If not :

- If you want to export your conda environment dependencies : `conda list -e > requirements.txt`
- If you want to export you current python.exe dependency (will export conda deps if you are in a conda env) : `pip freeze > requirements.txt` or `pip3 freeze > requirements.txt`

🗃 Explicit every necessary packages !

You can manually add external url to install, the `pip install -r requirements.txt` will just execute `pip install` while appending the next line of the file. so a url like `git+ssh://git@github.com/flccrakers/moleculecrawlers#egg=moleculecrawlers` can be added too !

Code re-usability

❓ Why

At NovAliX, we want to write as little code as possible for as much functionality as possible (who doesn't).

We also have an increasing demand for specialized tool sets for our own usage and efficiency. With the multiplication of projects and team members, it will be increasingly difficult to not write some features two times in separate implementations.

Thus, we should always try to write our code with re-usability in mind.

Here are some Guidelines/Resources for how to develop in the most re-usable, and composable friendly way possible

👍 Humble Reminders

■ Write code for others as well as you

When you implement some heavy algorithm that could be used for other purpose or even simply easily re-usable, try to separate it(decouple) from the business it's linked to.

If there is more than one algorithm at play, put them in a specific module !

That way the logic will be re-usable in the rest of your application but also, maybe later, put in a published module for other projects to use !

■ Name things in a way that can be understood by somebody not in the same project as you

Because people need to be able to browse our internal library and find if some code or algorithm hasn't been done before.

■ Document the non-trivial behavior of your module ! And simply the API you are developing.

What good is a great module for SDF browsing if you have no idea how to use it ? A `README.md` file can bring a great deal of information and !warning! about the usage, limitations, or constraints of your module.

■ Use standardized objects to document in-code the data flow of your module

- Use `Classes` and `Types` to describe inputs and outputs of your functions
- Use `DTO` if there are external transmissions
- Use a `config.json` (or anything else) for standalone projects and environment specifics variables (Examples)
- Use a `version.txt` (or similar) to track your project changes, it's very helpful when working with other people ! (Examples)

🛠️ Some technical and theoretical tools aimed for re-usability

- Interfaces and abstract classes (C#)
- Web Components (HTML/JS)
- Pure Components (React)
- Data Classes (Python)
- Modules (JS / Python (mainly but really all modern languages))
- SOLID principles (Theory of POO Reusability)

🧱 Concrete implementation of re-usability at NovAliX

Status	Concept
ONGOING	Microservices approach to projects
TODO	Python module deployed as DevOps artifacts
TODO	Web Component as DevOps artifacts
TODO	React NovAlix Library

Indexes on database

Advanced PostgreSQL indexing tips in Django | Idego Group

Quick resume of most common indexes, not really exhaustive :

Hash Index – Summary:

- Reduces index size.
- In specific scenarios, it optimizes speed.
- Great for big-size, almost unique values.
- Inefficient when column data consists of similar values.
- Does not allow for index sorting, composite and unique values or range search.

```
from django.contrib.postgres.indexes import HashIndex


class Movie(models.Model):
    # ...
    class Meta:
        # ...
        indexes = (
            HashIndex(
                fields=('original_title',),
                name="%(app_label)s_%(class)s_original_title_ix",
            ),
            # ...
        )
```

Partial Index – Summary:

- Great for reducing index size.
- Does not affect query speed.
- Great for nullable columns.
- Most efficient when a column is limited by its values set.

```
from django.db.models import Index
```

```
class Movie(models.Model):
    # ...

    class Meta:
        # ...
        indexes = (
            models.Index(
                fields=("title_type",),
                name="%(app_label)s_%(class)s_t_type_movie_ix",
                condition=Q(title_type='movie')
            ),
        )
```

Conventionnal naming in project

Back-end Convention

Python and Django convention :

Where ?	Element	Convention	Example
Explorer	Module	lowercase_with_underscores	`my_module.py`
Explorer	Django Service	lowercase_with_underscores+'service'	`"service_name"_service`
Explorer	Django Controller	lowercase_with_underscores+'controller'	`"controller_name"_controller`
Explorer	Django Serializer File	lowercase_with_underscores+'DTO'	`"serializer_name"DTO`
Code	Django Serializer Class	CapitalizedWords+'DTOSerializer'	`"SerializerName"DTOSerializer`
Code	Django Model Class	CapitalizedWords	`"ModelName"Model`
Code	Global Variable	lowercase_with_underscores	`my_global_variable`
Code	Exception	CapitalizedError	`MyError`
Code	Private attribute/method	_single_leading_underscore	`_private_var`
Code	Protected Variable	single_trailing_underscore_	`protected_var_`
Code	Magic Method	__double_leading_underscore__	`__init__`
Code	Class Attribute	lowercase_with_underscores	`class_attribute`
Code	Django Field/Column	lowercase_with_underscores	`my_field`
Code	Django ForeignKey	lowercase_singular_model	`related_model`

Front-end Convention

Category	Naming Convention	Examples
Components	PascalCase	Header, UserProfile, LoginForm
Files and Folders	PascalCase for files, lowercase-hyphen for folders	Header.js, UserProfile.jsx, login-form.js
Props	camelCase	title, userInfo, onSubmit, imageUrl
State/Variables	camelCase	isLoading, errorMessage, userData, selectedItem
CSS/Styles	lowercase-hyphen for classes, lowercase-hyphen or underscore for files	.header, .user-profile, styles.css, styles.module.css, .button, .button--primary
Function to render a part of a component inside a component function	“render” + camelCase	renderRow, renderAvatar …
Event Handlers in component functions	“handle” + camelCase	handleClick, handleChange, handleSubmit
Test a state / return a boolean	[can/is/do/are/…] + testedState in camelCase The function must ask a question	isUserOnline, areRowsPresent, canUserDoThat

How to win at Microservices

⚠️ Mandatory read if your project API will be called by another app.

ℹ️ If you are the one consuming another application's API, you have the right to request that the other project follow those guidelines.