

**UL DEPLOY**

Handbook

June 2017



**Internal Use Only**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Update Initiator** | **Date** | **Update Description** |
| 1.0 | Jonathan Berdah | June 2017 | Initial Version |
|  |  |  |  |

|  |
| --- |
| TABLE OF CONTENTS |

[**TABLE OF CONTENTS**](#_rfmruw5cmohm) **2**

[**INTRODUCTION**](#_6dmqq2lk86cp) **4**

[Purpose of This Document](#_xtblyt56n1ak) 4

[Definitions](#_vuj3xbp4y4yu) 4

[What is UL DEPLOY?](#_v6bu9yl2af7e) 5

[UL DEPLOY Tools](#_y0fhxs2q3bk) 6

[UL DEPLOY Architecture](#_sbkbmy7pboog) 7

[**KEY CONCEPTS**](#_n79rpji14a3y) **8**

[Packaged Approach](#_r0s8n42po5qu) 8

[Environments Are Now Really Environments](#_c203qg88uou0) 8

[Source Control for UL BRIDGE and UL ODISYS](#_6iw3w2ytki8l) 9

[UL BRIDGE](#_yv4pdm4skz59) 10

[UL ODISYS](#_sgxa32b8xb3) 11

[**TYPICAL WORKFLOW**](#_5h2jc460vmk8) **12**

[Scenario](#_ionn045c3u14) 12

[Using Git Branches](#_9gt99els1vs3) 12

[Part 1 - CTB Workflow](#_rgkdhkgdazue) 13

[Phase 1:   
UL PROD Initiates the Platform in UL DEPLOY](#_ghbvlcs27mr6) 13

[Phase 2:   
CTB & FIX Teams Work on the First Release Going into the Production Environment](#_cxpqbxgrzstt) 14

[Phase 2.1 - Cloning the DEVOP-PA\_ULBRIDGE Platform](#_sjg57gfvcgwe) 15

[Phase 2.2 - Creating a Branch and Performing a Change](#_l3erava9q7p0) 17

[Phase 2.3 - Pushing to Gerrit for Review and Package Building](#_68bo56qykgb1) 21

[Phase 2.4 - Releasing with XL DEPLOY](#_jg1dnuehz0w5) 24

[Phase 2.5 - Preparing the next-release Branch](#_lmdos9u4vaxg) 27

[Merging the CTB Feature Branch into next-release](#_l5jra8w32j97) 28

[Merging the FIX Feature Branch into next-release](#_5h987rrzgxb7) 30

[Integrating the the FIX and CTB Features Together](#_grnoxrd0zy08) 31

[Publishing your Commits From Your Local Branch to the Remote Branch](#_y7r8xjx9vrxk) 32

[Phase 2.6 - Creating a Production-eligible Package](#_yoa022jzy8nl) 35

[Phase 2.7 - Generating the UAT and PROD RFCs](#_m1jzj4gdubh) 38

[Using the ul-rfc-bot on Slack](#_51fa39cgfba1) 38

[Generating the UAT RFC](#_o2ihxbhalw2b) 38

[Generating the PROD RFC](#_5aiqfss0gnmc) 39

[Phase 3:   
CTB & FIX Teams Continue Working on the Implementation & Perform a Hot-Patch](#_tcpvxtx2w2z5) 40

[Performing a Hot-Patch](#_6f6v6l48kml7) 41

[Part 2 - RTB Workflow](#_krovpuy9k6oe) 42

[Phase 4:   
The Platform is Live, RTB Performs Bug Fixes as New Features are Implemented](#_wzub76hkh0mm) 43

[Phase 5:   
New Bugs are Fixed, a Feature is Cherry-Picked from the Implementation Branch](#_pv1xm4xbs695) 44

[**APPENDIX**](#_k6spi7a89vv9) **46**

[Amend Commit](#_9inlqkc0zm5a) 46

[Parallel Commits](#_oa652wp93zsy) 47

[Deployment Instructions](#_m7eoo6p47icr) 49

[Database Management](#_y5wst5ehwr3j) 50

[Hot/Cold Deployment](#_pf9h1rill8a9) 53

[**FOR MORE INFORMATION…**](#_rgjwrfol4h79) **55**

[Contact](#_9aw4pwegbjm6) 55

[Other Documentation](#_z4z5uipbl4kt) 55

[Git Documentation](#_7vj9h4fofnfr) 55

[Gerrit Documentation](#_ayd9adtbjz1o) 55

[XL DEPLOY Documentation](#_lunyu6t7v29l) 55

# 

# 

# 

# 

# INTRODUCTION

## Purpose of This Document

This document is an introduction to UL DEPLOY, Ullink’s new deployment process which will be implemented worldwide in the Ullink Operations teams in 2017.

It gives an overview of UL DEPLOY and explains its principles through a detailed example of CTB/RTB workflow.

Links are provided at the end of this guide for more detailed information and tutorials on the tools used with UL DEPLOY.

|  |  |
| --- | --- |
| **NOTE** | This guide assumes that you are familiar with Ullink products and client platforms. |

## Definitions

The following terms will be used throughout this document:

|  |  |
| --- | --- |
| **Platform** | A single UL BRIDGE or UL ODISYS. |
| **Package** | A **.dar** file (like a zip file) containing a snapshot of the entire platform configuration, used by XL DEPLOY. |

## 

## 

## What is UL DEPLOY?

UL DEPLOY is a **process** that aims at replacing the way Ullink formerly handled the **deployment of clients’ platforms**, which was a sequence of manual, time-consuming and risky operations.

UL DEPLOY will be used for our clients’ entire project implementation life cycle, from their first platform implementation (by CTB Team) to their daily project life (CTB, RTB, Fast Track teams...).

|  |  |
| --- | --- |
| **Former Deployment Process** | **With UL DEPLOY** |
| Everyone is implementing on **a shared DEV environment**: overwriting changes, risk of errors when moving from DEV to UAT. | Everyone can implement **on their local computer**: no overwriting, implemented functionality works the same way when moved from DEV to UAT. |
| Working **without Source Control**:  Need to remember everything you’ve done to be able to move to the next environment (UAT, PROD), no history. | Working **with Source Control:**  Full tracking of the changes (who, when, what) with branching, merging, versioning... |
| Implementing a change requires **writing down instructions** to modify an existing platform (delta): complex, time-consuming, leads to potential errors during description/execution. | Packaged approach: what is delivered to the next environment is a **fully-packaged application** instead of change instructions.  Changes are deployed in UAT and can be tested by clients, then undeployed if necessary (but stay in Source Control with no work loss). |
| Executing changes in UAT and PROD is a **manual process** done by UL PROD team. | UL DEPLOY allows an **automated and easy deployment** in each environment with  XL DEPLOY.  CTB is allowed to deploy in UAT without the  UL PROD team. |

|  |  |
| --- | --- |
| **NOTE** | For more information, see:   * [UL DEPLOY Tools](#_y0fhxs2q3bk) * [UL DEPLOY Architecture](#_sbkbmy7pboog) * [KEY CONCEPTS](#_n79rpji14a3y) |

## 

## 

## UL DEPLOY Tools

With the UL DEPLOY process you will be using the following tools:

|  |  |
| --- | --- |
| **GIT** | Centralizes and versions all changes performed to a platform.  Automatically detects conflicts and forces users to deal with them.  [Video tutorial of Git + Git extensions](https://www.youtube.com/watch?v=cFbCusX9bKs) |
| **GERRIT** | Key people (architect/PROD team/senior colleagues) review changes before delivery to customer. |
| **UL CLONE** | Internal tool allowing to easily replicate a client platform on an end-user’s computer, to work on an implementation or in order to reproduce an issue.  Standard installation for UL DEPLOY can be found [here](https://sites.google.com/a/ullink.com/prod/uldeploy/desktop-setup). |
| **JENKINS &**  **XL DEPLOY** | Every commit to the Git repository is turned into a package that can be automatically deployed to DEV > UAT > PROD. |
| **UL TEST** | Automated tests for each new feature brought to a platform, allowing for future non-regression testing (better quality in our deliveries). |

|  |  |
| --- | --- |
| **NOTE** | To see how these tools interact with each other, see: [UL DEPLOY Architecture](#_sbkbmy7pboog). |

## UL DEPLOY Architecture

### 

|  |  |
| --- | --- |
| **1** | On a user computer, UL CLONE performs a copy (aka “checkout”) of the platform’s Git project. |
| **2** | UL CLONE then performs an RSYNC of the platform’s daily backup of the PROD environment: files are copied on top of the ones retrieved during the previous step (thus allowing to retrieve startup scripts, CSVs which are not under source control, etc.). |
| **3** | The user can now locally work on the platform. When happy with the changes made on the platform, the user pushes them to Gerrit. |
| **4** | With Gerrit, the changes are reviewed (Review bot + Peer review). |
| **5** | Once the changes are validated (“+2” review) and submitted, Jenkins automatically builds a package of the platform. |
| **6** | Jenkins pushes the package to XL DEPLOY. |
| **7** | The package can now be deployed to DEV/UAT/PROD. |

# KEY CONCEPTS

## Packaged Approach

**Before UL DEPLOY, performing a change on a client’s platform consisted in a series of manual operations:**

* Manipulating the DEV environment,
* Remembering all modified/added/deleted files and entries,
* Then creating RFCs to promote this delta all the way to UAT and finally to PROD.

**With UL DEPLOY, a new package can be created for each change:**

* What will be deployed in DEV, UAT and PROD is a **full package of the application**   
  (UL BRIDGE/UL ODISYS).
* Even a small change in a .java, .cfb, .ini or .csv file will end up in a new built package, if this small change needs to go by itself to Production.
* Only the changes related to the data will continue to go through the legacy RFC process: creating/modifying a user on UL ODISYS, deleting/dumping/updating instruments, changing the default extension of a specific user, etc.
* If a file is managed by the client/3rd party (uploaded through SFTP), it has to be taken out of the source control (and thus the package).

## Environments Are Now Really Environments

**Before UL DEPLOY, each environment was a “version” of the application:**

We could consider that PROD was “V1”, UAT “V2” (PROD + new features & bug fixes) and DEV was “V3” (UAT + new features & bug fixes).

There was always a risk to delete UAT or DEV content as they might contain features not existing anywhere else (besides JIRA).

**With UL DEPLOY, DEV and UAT are no longer the place where a change is stored:**

DEV and UAT are now “real” environments in which anyone can deploy a specific package of   
UL BRIDGE or UL ODISYS.

At any given time, DEV and UAT can have the same package as the one currently deployed in PROD, or another one with a specific feature, or another one with a set of bug fixes, etc.

## Source Control for UL BRIDGE and UL ODISYS

Each UL BRIDGE and UL ODISYS has its own project in Git, with its own history, branches, etc.

This Git project is initialized by the UL PROD team, when they create the first version of the platform, based on a template they have put together.

The idea is to **track the whole configuration** of a UL BRIDGE/UL ODISYS.

This means that we want to commit **the entire folder** containing the application, **without**:

* **Binaries**: there is no point putting binaries under source control. They are all available on each physical server in our datacenter and synchronized daily. They are referenced by the configuration files.
* **Startup scripts**, and other **.sh** files: these files are under the control of UL PROD team. They can be updated without releasing a new version of the application.
* **Logs**: they have their own life.
* **Data**: there is no point putting the database under source control. It has it’s own life and changes every second!

|  |
| --- |
| Ultimately, this is what should be under source control for a UL BRIDGE/UL ODISYS:   * [**UL BRIDGE Folders Under Source Control**](#_yv4pdm4skz59) * [**UL ODISYS Folders Under Source Control**](#_sgxa32b8xb3) |

### 

### 

### UL BRIDGE

|  |  |  |
| --- | --- | --- |
| **Folder** | **SubFolder/File Under Source Control** | **Explanation** |
| **/bin** | **client-ulprod.parameters** | This file is created by the UL PROD team when they initialize the platform, and it stays under their management.  It uses specific hard-coded values (like the SERVICE\_NAME) but also variables, injected by  XL DEPLOY at deployment time, related to the environment (JVM Version, Heap/Young/Perm Size, etc.). |
| **<dev/uat/prd>-<platform\_name>.properties** | These 3 files are initiated by the UL PROD team but maintained by the CTB/RTB/FT/FO teams.  They contain all variables used by the configurations files (hot failover specifics, IPs, ports, SCIs, TCIs, etc.) and the plugins/modules/market data adapter versions used by the platform, for DEV, UAT and PROD environments. |
| **jmx-client.properties** | This file is created by the UL PROD team when they initialize the platform. It shouldn’t change during the life of the project. |
| **version.parameters** | Contains the CORE Product versions. |
| **/conf** | **\*** | This whole folder is under source control.  If the client has the capacity to change some CFBs/enrichments through SFTP upload, these files need to be specifically added to the **.gitignore** file, so that they won’t be under source control anymore. |
| **/lib** | **/custom\_libraries/\*** | If any custom library is needed on the platform (jdbc driver, specific library for enrichment, etc.), this is the folder in which it should be added in the source control.  There should be nothing else in the **/lib** folder in Git. |
| **/manualOps** | **deployment-instructions.txt** | This file is used to keep track of all manual operations that need to be performed for a specific release (license update, JMX calls...). |
| **/tests/** | **\*** | This folder contains the tests associated to the platform.  More about it here. |

### 

### 

### UL ODISYS

|  |  |  |
| --- | --- | --- |
| **Folder** | **SubFolder/File Under Source Control** | **Explanation** |
| **/bin** | **client-ulprod.parameters** | This file is created by the UL PROD team when they initialize the platform, and it stays under their management.  It uses specific hard-coded values (like the SERVICE\_NAME) but also variables, injected by  XL DEPLOY at deployment time, related to the environment (JVM Version, Heap/Young/Perm Size, etc.). |
| **<dev/uat/prd>-<platform\_name>.properties** | These 3 files are initiated by the UL PROD team but maintained by the CTB/RTB/FT/FO teams.  They contain all variables used by the configurations files (hot failover specifics, ip, ports, etc.) and the modules/market data adapter versions used by the platform, for DEV, UAT and PROD environments. |
| **jmx-client.properties** | This file is created by the UL PROD team when they initialize the platform. It shouldn’t change during the life of the project. |
| **version.parameters** | Contains the CORE Product versions (UL ODISYS version, extensions). |
| **/conf** | **\*** | This whole folder is under source control. |
|  | **/ul-desk/users/<dev/uat/prd>-users-template-settings.csv** | If the market data configuration is templated, these 3 files should be created (one for each environment), with this specific naming convention. |
| **/lib** | **/custom\_libraries/\*** | If any custom library is needed on the platform (jdbc driver, specific library for enrichment, etc.), this is the folder in which it should be added in the source control.  There should be nothing else in the **/lib** folder in Git. |
| **/manualOps** | **deployment-instructions.txt** | This file is used to keep track of all manual operations that need to be performed for a specific release (user creation, license update, JMX calls, XML/FIXatdl to upload, trigger import, etc.). |

# TYPICAL WORKFLOW

## Scenario

The CTB and RTB workflows belows describe a typical new platform implementation and life cycle:

**Part 1 - Pre-production (CTB):** First, the UL PROD team creates a platform from scratch in which the CTB & FIX teams implement plugins, modules, enrichments…   
See [Part 1 - CTB Workflow](#_rgkdhkgdazue).

**Part 2 - Day-to-day life (RTB):** Then, once the platform is pushed to production, it is handed over to the RTB team who is now in charge of running the day-to-day issues: bug fixes, client requests... Some features will also be implemented during this part of the platform’s life.   
See [Part 2 - RTB Workflow](#_krovpuy9k6oe).

## Using Git Branches

In the diagrams used to illustrate this workflow, besides the Time, each arrow represents a branch which will be used in Git during the project life (pre-production and during production):

* **master**: the **master** branch is here by default in Git. Only a commit done in that branch will generate a package eligible to PROD. Anyone can commit changes to this branch but the peer review will always be done by the UL PROD team. Committing into this branch basically only makes sense when the decision is made to move a set of tested and validated features/bug-fixes to PROD.
* **next-release**: the **next-release** branch is used to integrate the **feature**/**bug-fixes** branches and ensuring that everything works well together before going to the **master** branch. Clients can test packages built from commits in this branch in a very dynamic way: the **next-release** packages can be deployed in DEV and UAT by everyone, no need to involve the UL PROD team (unless some data manipulation is required).
* **Feature Branch** (CTB, FIX Onboarding): these branches will not necessarily be named “feature branch”. They represent branches that will be created by the CTB and FIX Onboarding teams when working on features to implement (e.g. creating a plugin or implementing an extension). Feature branches can be deployed in DEV and UAT by everyone, but they don’t have to be! Once a feature is considered as part of the next package going to PROD, the package can be merged to the **next-release** branch.
* **bug-fixes**: this branch is used to fix bugs and perform small changes once the platform is in PROD. The CSM will always try to merge this branch as a whole inside **next-release**, meaning that the client has to validate all the bug fixes in it and not pick and choose some (which is different from the features, which are split over different branches).

## Part 1 - CTB Workflow

In this part of the scenario we’ll describe an example of pre-production workflow for a new platform implementation.

It has been divided into 3 phases:

* [**Phase 1**](#_ghbvlcs27mr6): UL PROD initiates the platform in UL DEPLOY.
* [**Phase 2**](#_cxpqbxgrzstt): CTB & FIX Onboarding teams work on the first release going into the Production environment (pre-production, not handed over to RTB yet).
* [**Phase 3**](#_tcpvxtx2w2z5): CTB & FIX Onboarding teams continue working on the implementation & perform a hot-patch.

|  |
| --- |
| Phase 1:  UL PROD Initiates the Platform in UL DEPLOY |



In this first phase, the UL PROD team initiates the platform on UL DEPLOY.

They will follow [this documentation](https://sites.google.com/a/ullink.com/prod/uldeploy/xl-deploy/initiate), in order to:

* + Initiate the Git Project.
  + Create the Jenkins jobs (required for automatic build of the packages).
  + Initiate the platform in XL DEPLOY (creation of the application, declaration of the environments, the VIPs to access them, the pipeline…).
  + Create the 0.0.0 version of the application, based on a UL BRIDGE/UL ODISYS template.
  + Commit the content of the 0.0.0 package in Git.
  + Deploy the 0.0.0 in DEV, UAT & PROD.
  + Usually, UL PROD will adjust some configuration, ending up creating a 1.1.1 that will be deployed in DEV, UAT & PROD as well before informing the CTB that the platform is ready for them.

|  |
| --- |
| Phase 2:  CTB & FIX Teams Work on the First Release Going into the Production Environment |

#### 

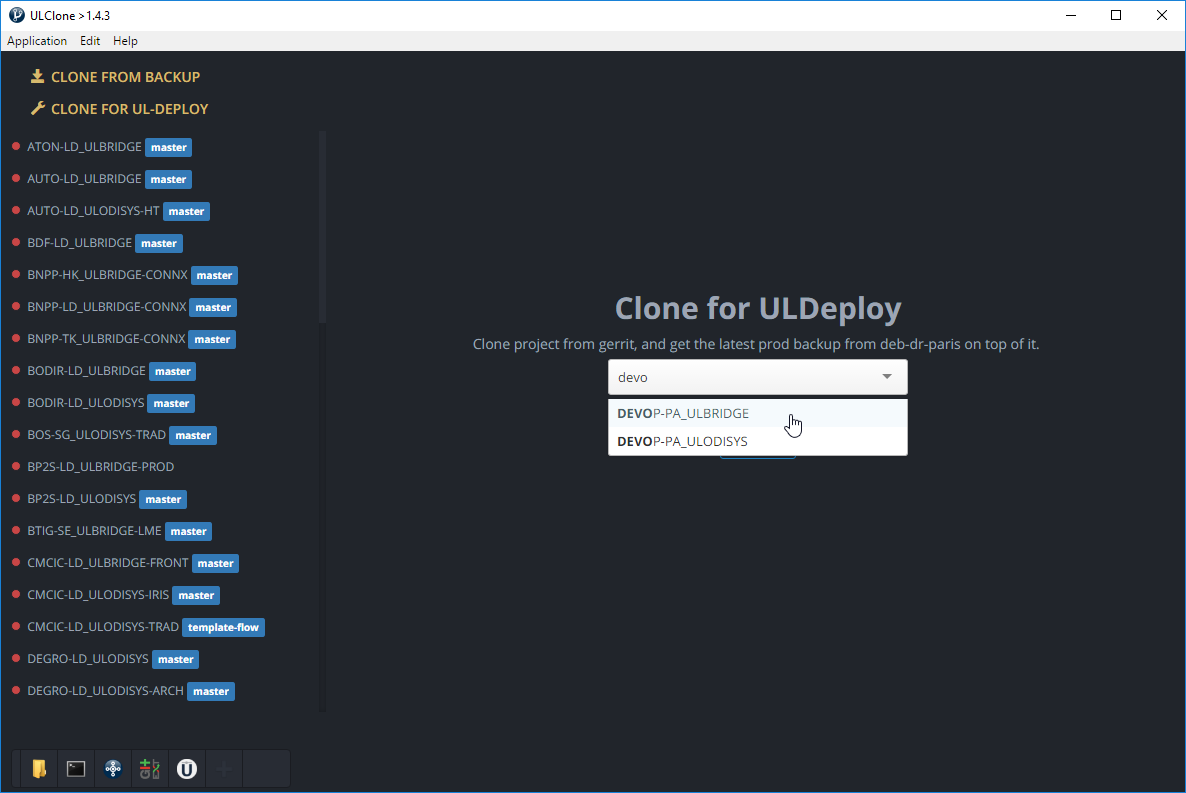
In this second phase, once the platform is initiated by the UL PROD team, CTB & FIX Onboarding teams can start implementing. In this scenario:

* Two plugins (inbound and outbound) will be created on a **Feature Branch** managed by a FIX Onboarder.
* UL FLOW will be implemented along with the instrument resolution on another **Feature Branch** managed by a CTB Engineer.
* All these features will be merged into the **next-release** branch until they are validated to become a full package on the **master** branch.
* This package will be a deployable version of the platform in Production.

|  |  |
| --- | --- |
| **NOTE** | In this scenario, the platform name is **DEVOP-PA\_ULBRIDGE**. |

##### Phase 2.1 - Cloning the DEVOP-PA\_ULBRIDGE Platform

1. Launch UL CLONE.
2. Select **Clone for ULDeploy**, then select **DEVOP-PA\_ULBRIDGE** in the drop-down menu:



This completes steps **1** and **2** of our architecture:

1. Cloning of the platform from the repository and positioning it on the commit with the **PROD** tag,
2. RSYNC of the Production files from DEB-DR-PARIS.

### 

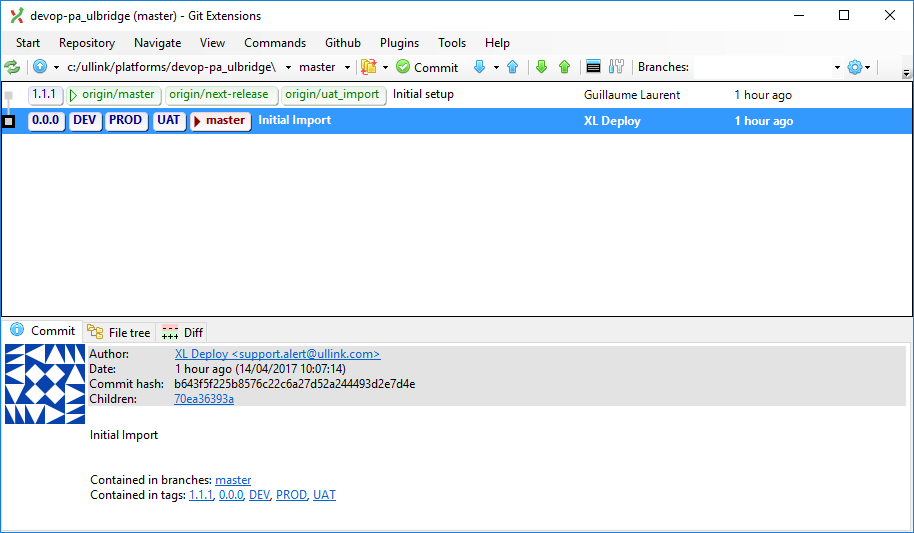
UL CLONE will parse the **version.parameters** file and the **prd-devop-pa\_ulbridge.properties**, and then:

* RSYNC all the required binaries from **deb-ulprod-repo-mirror.ullink.lan**.
* Generate a **prd-devop-pa\_ulbridge.properties.windows** file, which is going to be used to start the platform locally. All the linux path have been replaced by Windows ones, and the IPs are replaced by **localhost**.

|  |  |
| --- | --- |
| **NOTE** | RSYNC of binaries and generation of the **prd-devop-pa\_ulbridge.properties.windows** will be performed every time you start the product through UL CLONE. |

Once the cloning process is over, you can edit **client.parameters** to replace the HEAP in order to be able to start the platform on your local computer. This file is originally coming from DEB-DR-PARIS and is not under source control (it is ignored by Git): even if you modify it, it will never be committed.

Now that the platform is cloned (and should be able to start!), you can open the Git Extensions to have a look at the Git history and start working:



##### 

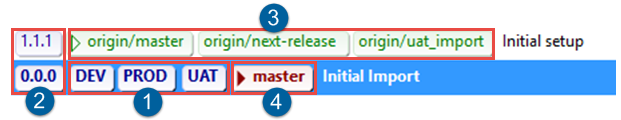
##### 

##### Phase 2.2 - Creating a Branch and Performing a Change

By default, only the **master** branch from Git is present. It cannot be deleted.

The UL PROD team should have made 2 (or more) commits in that branch: one with a **0.0.0** tag, one with a **1.1.1** tag (and more if more commits were made).

The cloning process “checked out” the **master** branch on your computer, creating a “local” **master** branch. It then moved to the commit which has the **PROD** tag. This means that currently, on your computer, the files in **C:\Ullink\Platforms\DEVOP-PA\_ULBRIDGE** are those of that specific commit.



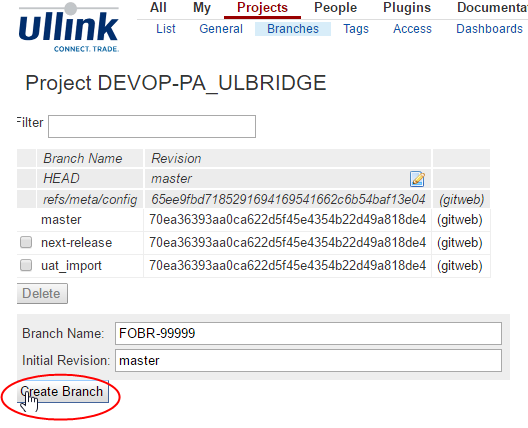
|  |  |
| --- | --- |
| **1** | **Environment tags:** Indicates which commit is currently deployed in the DEV, UAT and PROD environments. These 3 tags can obviously be on different commits! |
| **2** | **Version tags:** Version number of the package associated with the specifics commits. On the **master** branch, they always follow this model (0.0.0, 1.1.1, 1.2.1, etc.), whereas on other branches, they have the name of the branch followed by an incremented number (e.g.: next-release.35). |
| **3** | **Remote branches**: Indicates where remote branches (visible to everyone) currently are. |
| **4** | **Local branch:** Indicates where your current local branch is (checked out from the remote repository or created manually on your local computer). |

**Now let’s say that you are a FIX Onboarder**, and you would like to implement your first **FIX inbound and outbound plugins** as requested by the project manager in a **FOBR-99999** Ticket:



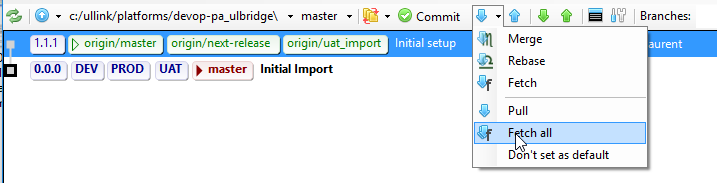
You do not want to work directly in the **master** branch, as committing in the **master** branch is only once you are ready to move your changes to Production.

Therefore, you create a new branch to work on, originating from the **master** branch, which is something you can do in Gerrit:

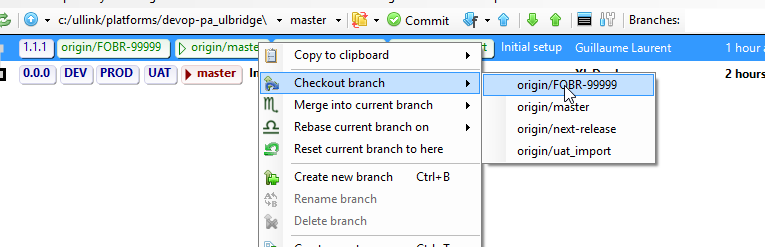


Once the branch is created, you must:

1. Refresh your Git Extensions with the **Fetch all** menu:



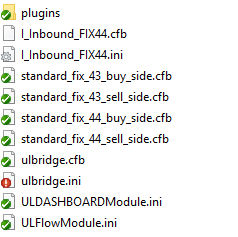
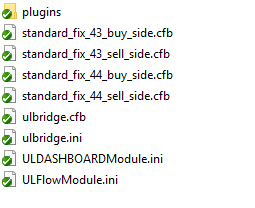
1. Checkout the **FOBR-99999** branch to start working on it:



You can now start working on the platform by either editing files through your favorite editors or through the Configuration Manager.

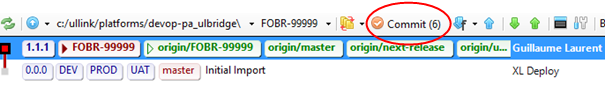
|  |  |
| --- | --- |
| **NOTE** | If you are using editors, we highly recommend the following:   * Eclipse/IntelliJ for Java enrichment edition * Visual Code for the rest |

Creating the first Inbound Plugin should result in:

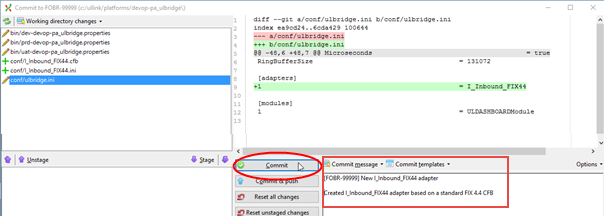
* **2 files created**: **I\_Inbound\_FIX44.ini** and **I\_Inbound\_FIX44.cfb** in our example.
* **4 files modified**: **ulbridge.ini**, **dev/uat/prd-devop-pa\_ulbridge.properties**.

|  |  |
| --- | --- |
| **NOTE** | **Always modify the 3 properties files in the same commit.**  Do this when you update a plugin/module version (should be aligned in the 3 files) or when you declare new properties for plugins (ip, port, etc.).  If you do not have a DEV/UAT session or you do not know the “real” values for PROD yet, define the ip to **127.0.0.1** and the port to **1800**. |

Git detects that 2 files were added and 4 files modified, hence the icons in your explorer and the **Commit (6)** button in the Git Extensions:



After testing this locally, and making sure it works as expected, you can now commit your change, with a commit message:

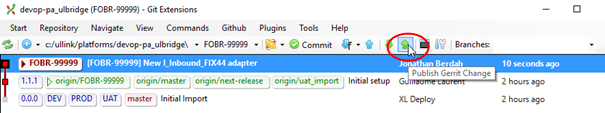


You have now committed this change in your local **FOBR-99999** branch.

Nobody besides you can see this. You could continue working locally, create another commit, then decide to rollback to this specific commit, etc.

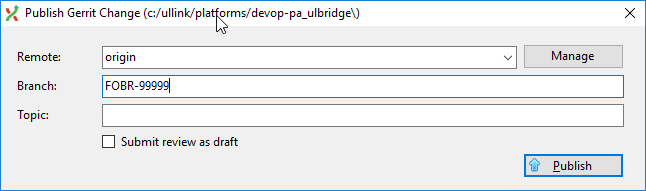
But you might want to verify that the IP, port, SCI and TCI you were provided with are correct and that the connection with the counterparty is up in UAT.

It is now time to push your change to Gerrit (do a **Fetch all** before, to retrieve potential changes done while you were working on your side), in order to build a DEV and UAT deployable package:



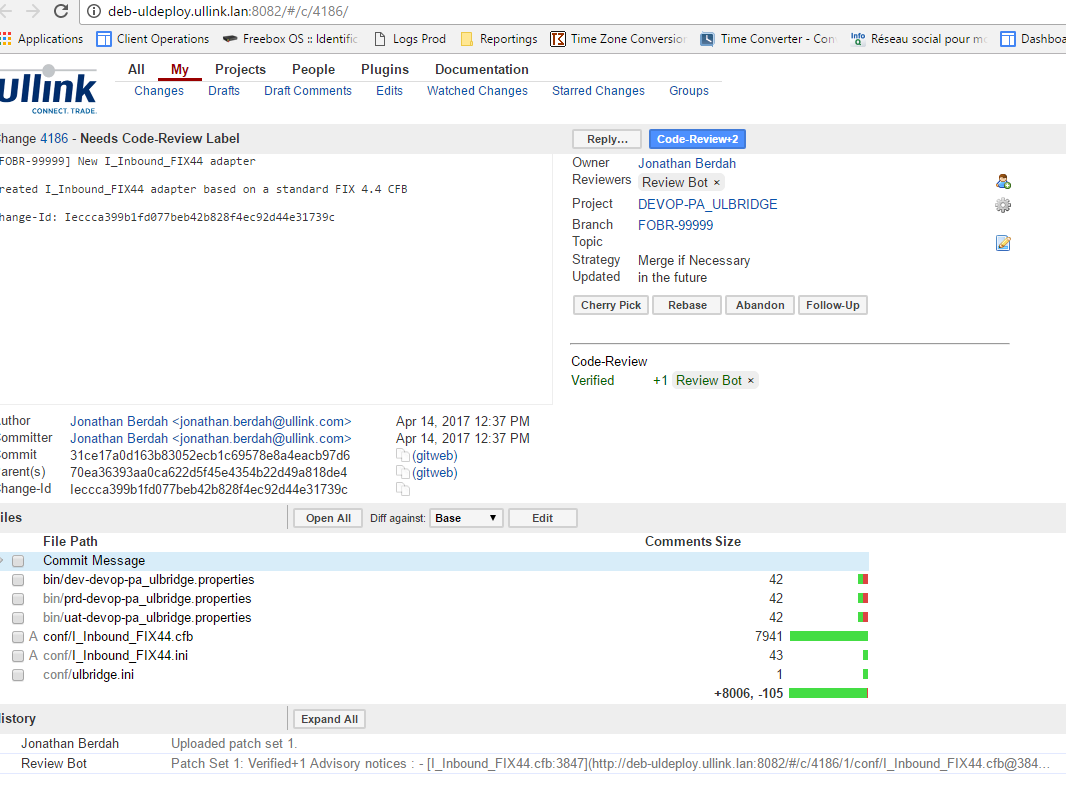
##### Phase 2.3 - Pushing to Gerrit for Review and Package Building

Once you have clicked the **Publish Gerrit Change** button, the following pop up opens:



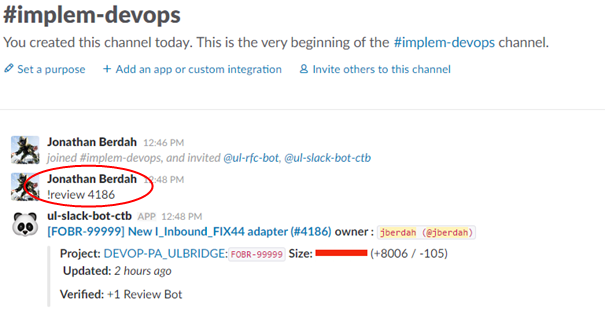
|  |  |
| --- | --- |
| **WARNING** | **Be careful before clicking Publish!**  The **master** branch is selected by default. You must cut/paste the **Topic** into the **Branch** field, unless you want to push to the **master** branch. |

The push provides a link to the Gerrit review page:



You must now:

1. Go to your dedicated **implementation-<clientname>** Slack channel.
2. Post: **!review <GERRIT\_REVIEW\_ID>**:

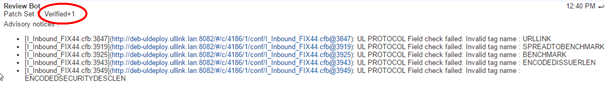


**Peer Review:**

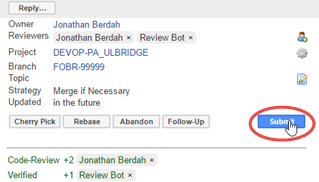
This post notifies everyone working on the project that a new commit has been made.

This allows your colleagues to review it and warn you if they see a mistake, or if something you have designed will be incompatible with something they are working on on their branch.

**Review Bot:**

In parallel, the review bot will analyze the 6 files you have modified and give you a **+1** or a **-1** in the **Verified** status:

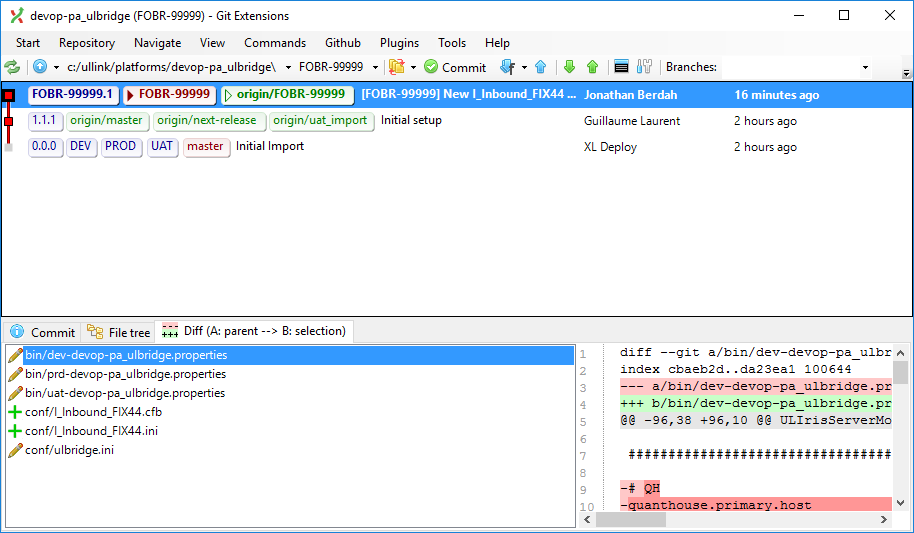
* **-1**: You cannot commit your code. You either need to fix your commit (see [Amend Commit](#_nby7wymz2waq) in Appendix) or ask an architect to bypass the review bot (if its decision was not justified).
* **+1**: You can now grant yourself a **+2** if you don’t need any peer review (this depends on your seniority and the organization of the project, which will be defined by the CTB Engineer in charge of the platform).  
  This allows you to **Submit** your source code:



Submitting your source code triggers the Jenkins job, which:

* Builds a package,
* Pushes it to XL DEPLOY,
* Pushes a tag to Git with the name of the release.

Performing a **Fetch all** will update the information in your Git Extensions:



##### 

##### 

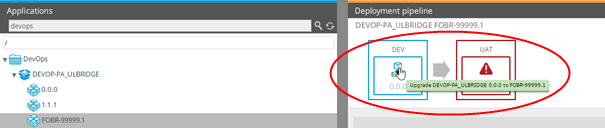
##### Phase 2.4 - Releasing with XL DEPLOY

Now that the package is available in XL DEPLOY, you can release it into the DEV environment. This is necessary before deploying it in UAT.

|  |  |
| --- | --- |
| **WARNING** | Before deploying anything, unless you are working alone on the platform, **always** **warn your colleagues** on your dedicated Slack implementation channel. |

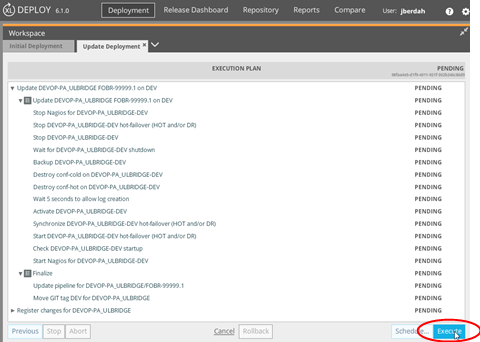
In XL DEPLOY:

1. Go into <http://xldeploy.ullink.net:4516>.
2. Select **Release Dashboard.**
3. Select your application (here, **DEVOP-PA\_ULBRIDGE**).
4. Select the package you want to deploy and click on the **DEV** environment in the right-hand side panel:



You can now either:

* Click **Preview** to see what XL DEPLOY is about to do,
* Or click **Advanced** if you want to skip some steps (should only used by experts !),
* Or click **Execute** if you are ready.



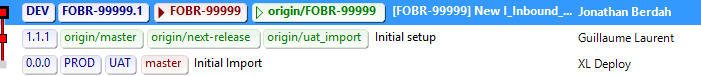
XL DEPLOY compares the previous deployed package (in this example, 0.0.0) and the one being deployed.

As this new package was built on a branch commit which is not the master one:

* This package will **never be allowed to be deployed in PROD** (XL DEPLOY will block it).
* A deployment in **DEV** or **UAT** will always **restart the UL BRIDGE/UL ODISYS**.   
  This allows you ensure that all your changes were taken into account, as you don’t have the rights to reload modules/plugins/enrichments/... in UAT, without involving the UL PROD team. This is not always true for package built from the master branch (see [Hot/Cold Deployment](#_jxnboltazbr2)).
* During the deployment process, there is a step that will pause right after the shutdown of the UL BRIDGE/UL ODISYS. It will ask you to click **Continue** before restarting: this allows you to perform manual operations such as restoring a database (see [Database Management](#_y5wst5ehwr3j)).

|  |  |
| --- | --- |
| **WARNING** | Please always be careful to click on the **“Close”** button at the end of the deployment through UL Deploy. |

At the end of the deployment, XL DEPLOY pushes the **DEV** tag to Git, letting everyone know that the DEV is now running on this commit. It also updates its pipeline, allowing you to deploy this package in UAT.



After warning your colleagues (and the client if they are already using UAT), you can now deploy in UAT.

If everything goes fine, it would now be the time to start working on the second part of your feature, which is the Outbound plugin:



|  |  |
| --- | --- |
| **NOTE** | Here, the two plugin commits are on the same branch, as they will move to PROD simultaneously. But they could also be on 2 different branches and merged later on. |

In the meantime, some CTB engineers worked on another **Feature Branch (CTB)**, and they have implemented UL FLOW and the ULLINK.INSTRUMENTID instrument resolution (along with some work in the associated UL ODISYS, but this won’t be covered here).

They have followed the same procedure, and possibly released their package in UAT at some point. As the **CTB Feature branch** does not have the same content as the **FIX Feature Branch**, deploying this package in UAT would result in an environment without the Inbound & Outbound plugins.

##### Phase 2.5 - Preparing the next-release Branch

It is now time to prepare the first release that will cover the scope of the first PROD release: one inbound plugin sending orders to an outbound plugin, with basic instrument resolution.

**Now, let’s say you are a CTB engineer**, in charge of creating the **next-release** branch, checking it out in the Git Extensions and taking care of the merge.

You will first merge the **CTB Feature Branch** into **next-release** (but you could merge the **FIX Feature Branch** first too).



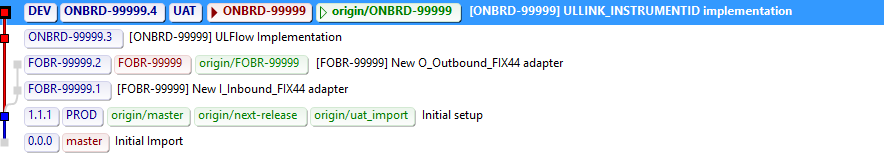
###### 

###### 

###### 

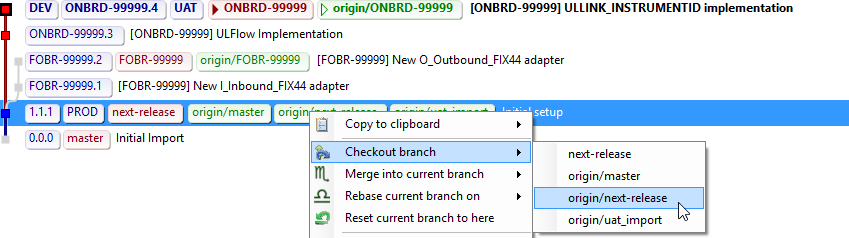
###### **Merging the CTB Feature Branch into next-release**

The situation in Git should be the following:

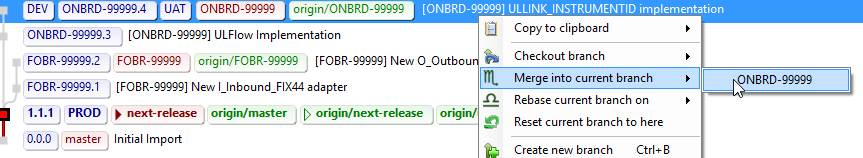


To merge the **CTB Feature Branch** into **next-release**, you must:

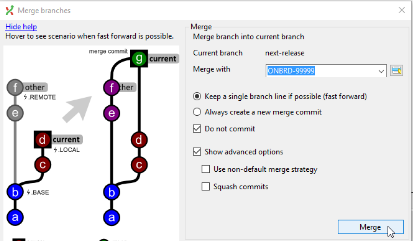
1. Check out **next-release** (which is currently at the same level as **master**):



1. Select the **ONBRD-99999** branch and merge it into **next-release**:



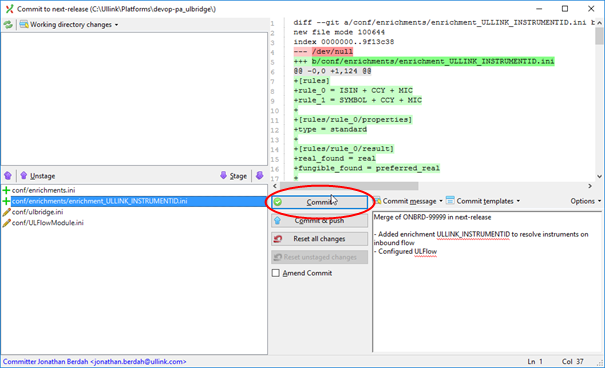
1. In the **Merge branches** window, you will select the following options:



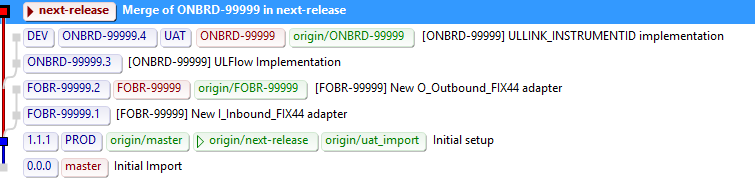
**Do not commit**: allows to review the commit and its message before committing it.

|  |  |
| --- | --- |
| **NOTE** | **When merging into the next-release branch you DO NOT have to check the “Squash Commit” option**. This way, you will always keep the visual link of the branches you have merged inside . |

1. You can now go back to the commit window and click **Commit**:



The merge has been made:



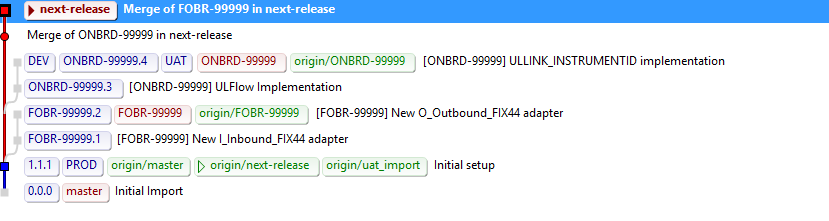
|  |  |
| --- | --- |
| **NOTE** | The screenshot above was done with the “Squash Commit” option activated: this led to losing the visual connection between the **ONBRD-99999** branch and the **next-release** branch. |

###### **Merging the FIX Feature Branch into next-release**

You now need to merge the **FIX Feature Branch** into **next-release** using the same method:



The result will be the following:



###### 

###### 

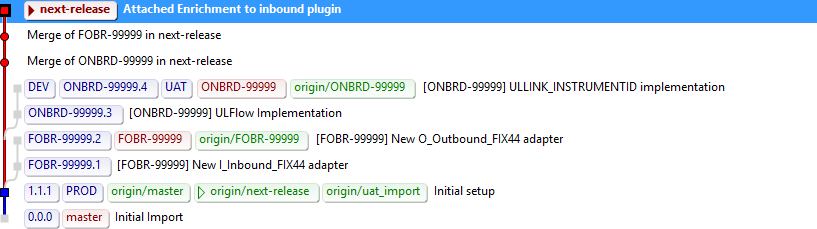
###### **Integrating the the FIX and CTB Features Together**

After merging the two **Feature Branches** into **next-release**, as a CTB Engineer you must now work on integrating them together.

It is now the time to attach the ULLINK.INSTRUMENTID to the inbound plugin. Once this done and tested on your local instance, you will perform a new commit on the **next-release** branch:



The result will be the following:



###### 

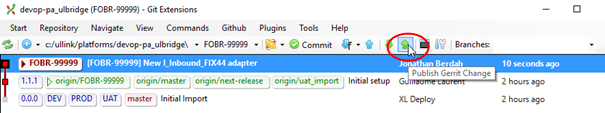
###### 

###### **Publishing your Commits From** Y**our Local Branch to the Remote Branch**

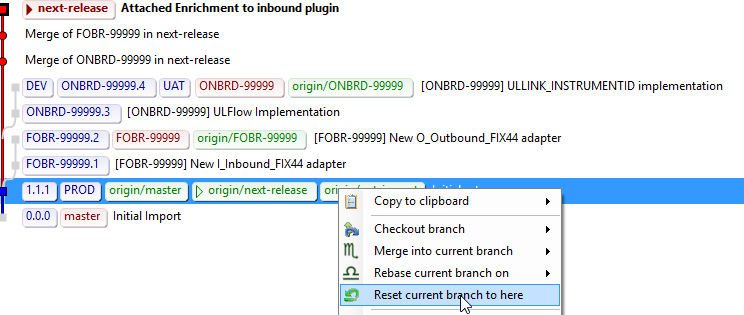
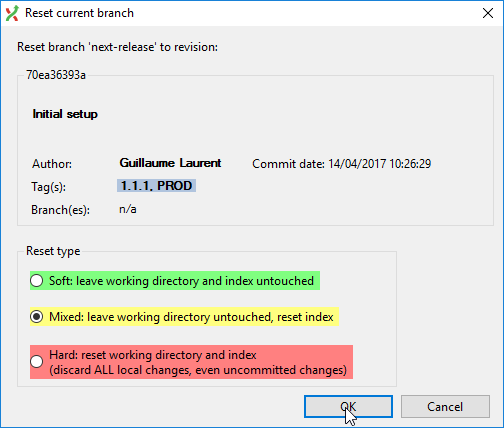
You now have 3 commits on your local **next-release** branch. You must publish this to Gerrit to update the remote branch and to get a package that you will be able to deploy in DEV and UAT.

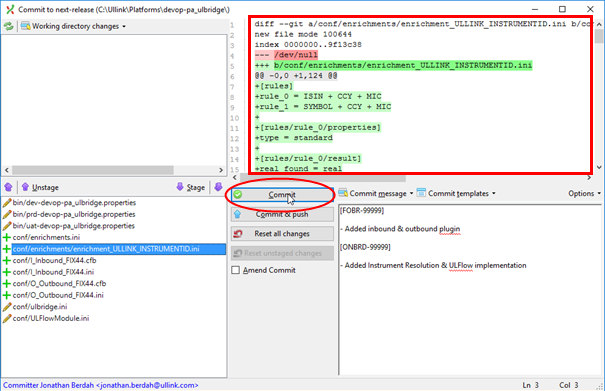
You have two options:

* Click the **Publish Gerrit Change** green arrow.  
  This will push the 3 commits as separate commits, which will all need to be reviewed and submitted and will generate one package per commit. (not recommended)

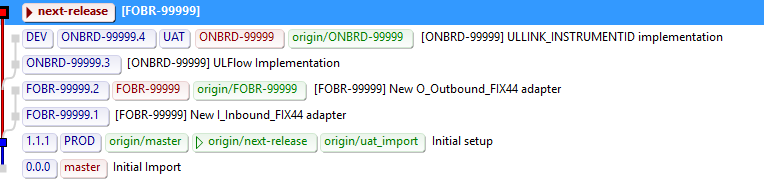


* Submit these 3 steps as a single commit (recommended)   
  To do so:

1. Reset your local **next-release** branch at the same level as the **origin/next-release** one:  
     
   
2. Choose the **Mixed** option in the pop-up window:  
   
3. Go to the **Commit** window and see all the changes between what everyone sees as **next-release** (the remote version of this branch) and the work you have done locally. You can now commit this as a single commit:



The result will be the following:



You can now publish that to Gerrit, using the same method as explained in [Phase 2.3](#_g5rkcmml97kd).

You will then be able to release this in DEV and UAT through XL DEPLOY and the client will be able to perform end-to-end testing between its two plugins with instrument resolution.

##### 

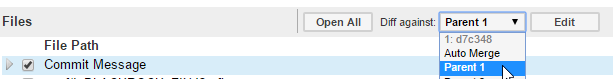
##### 

##### Phase 2.6 - Creating a Production-eligible Package

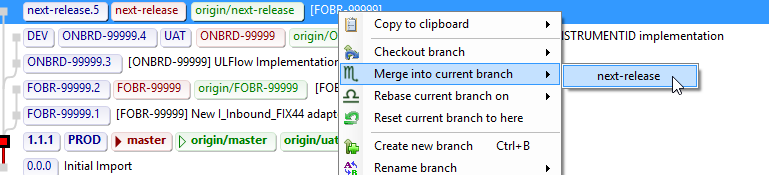
Once the client validates all the changes and is ready to move to PROD, it is time to merge **next-release** into the **master** branch, which will trigger the build of a Production-eligible package.

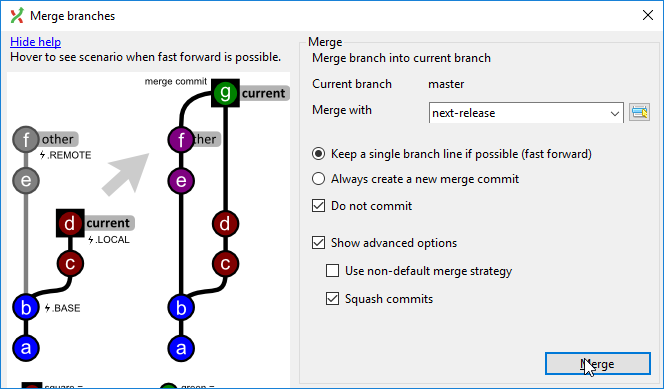
As a CTB Engineer, you must check out the **master** branch, and merge the **next-release** branch into it with the same options as previously (**Do not commit**),but this time also **Squash commits**:

* **Squash commits**: allows to create a single commit with the entire diff between the **next-release** branch and the **master**, and not a real “merge”. This will ease up the visualisation for the reviewer which will instantly see the touched files in Gerrit.   
  If you don’t use this option, the reviewer will have to select **Diff against Parent 1** in the Gerrit review to see the impact on the **master** branch:



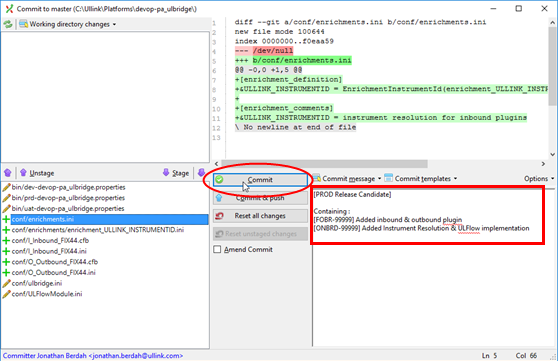
This should result in the following operation:



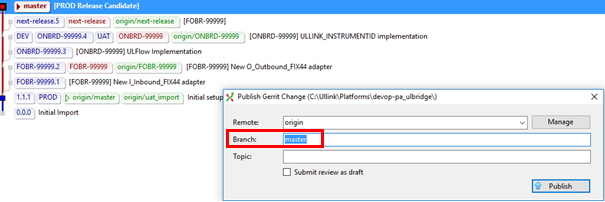


You must now commit this merge:

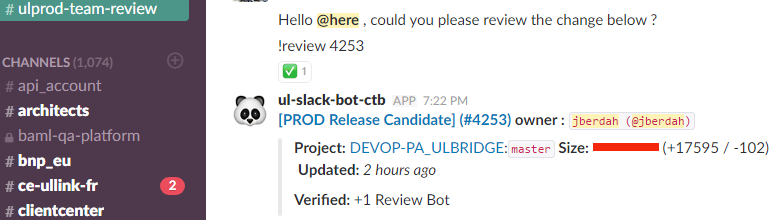
|  |  |
| --- | --- |
| **WARNING** | This time the **commit message becomes highly important**: it is what the client will see in their MCC later on as the “Business Reason”.  Please reference the SDK / ONBRD Jira tickets that this release is “closing” writing the line below at the end of your commit Message :  ***Issue: SDK-XXXXX, SDK-YYYYY, ONBRD-ZZZZZ***  This will :   * Allow anyone to easily use the search in Gerrit to find a commit linked to a specific ticket using *“tr:SDK-XXXXX”* in the Gerrit search * Automatically link these tickets to the RFC that you will generate with the rfc-bot (see below) |



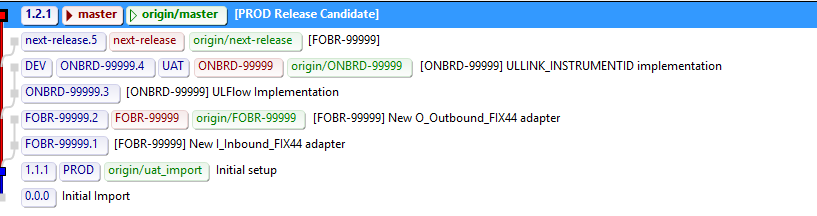
Once the merge is committed in your local master branch, you can push the change and this time you will do it in the **master** branch:



You must now get a **+2** from the UL PROD team, by asking a review on the **#ulprod-team-review** Slack channel:



Once the review is done by the UL PROD team, if your change is validated, you can see the following in your Git extensions:



|  |  |
| --- | --- |
| **NOTE** | If you got a **-1** or a **-2** , you must amend your commit. See [Amend Commit](#_nby7wymz2waq) in Appendix. |

You can now deploy this package in DEV and ask the UL PROD team to deploy it in UAT and PROD, as described in the next section.

##### Phase 2.7 - Generating the UAT and PROD RFCs

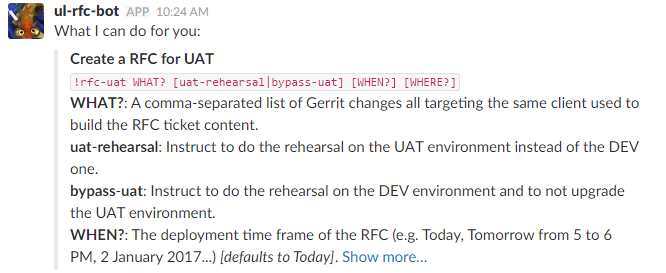
As a CTB engineer, now that you have a PROD-deliverable package, you might want to deliver it to PROD (you do not have to, but merging to **master** usually means you really want to deliver something in PROD!).

In order for the UL PROD team to feel comfortable with the release of this package in PROD, they are in charge of releasing it in UAT. This way they can:

* Validate the way XL DEPLOY will perform the upgrade of the platform.
* Test all the potential deployment instructions associated with the release (see [Deployment Instructions](#_259cgpb3qefp) in Appendix). Therefore, a UAT RFC must be created, preferably using the **ul-rfc-bot** on Slack.

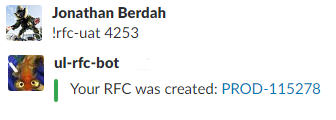
###### Using the ul-rfc-bot on Slack

This is what the Slack **ul-rfc-bot** can do for you:



###### Generating the UAT RFC

In the case of the DEVOP-PA\_ULBRIDGE release, as a CTB engineer you will go to the **#implem-devops** channel and enter:



The generated UAT RFC will instruct the UL PROD team to:

1. Perform a Dress Rehearsal in DEV, i.e.:
   1. Rollback DEV to the version currently running in PROD.
   2. Retrieve the PROD Database and put it in DEV.
   3. Deploy the 1.2.1 (associated to Git review 4253) in DEV.
2. Deploy the 1.2.1 (associated to Git review 4253) in UAT.
3. Perform all the manual instructions associated to the release.

The UL PROD team will decide if the Dress Rehearsal is necessary based on the difference between the current version running in PROD and the version about to be deployed.

They are more likely to perform it if:

* There is a version upgrade (plugin/module/extension/core product).
* There are deployment instructions with data manipulation.

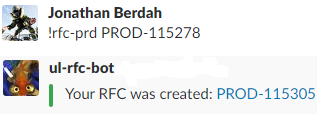
Two options can be used when creating the RFC:

* **uat-rehearsal**  
  Allows to request the Dress Rehearsal to be done in the UAT environment. This makes sense when the upgrade is massive (e.g. with data manipulation, product upgrades changing data schemas, etc.) and the client wants to make a full round of tests to validate the proper post-PROD data migration behavior.
* **bypass-uat**  
  Allows to request to NOT deploy the package in UAT and to flag it as deployable in PROD. This option can be used when the client platform is in “mix” mode (Project @CTB and BAU @RTB). Potentially, UAT is currently being used for the project, running on a feature branch with more advanced versions of products, new data, etc., and is being heavily tested by the client. In this case, a minor upgrade on the platform could bypass the UAT environment to avoid any disruption on the project.   
  Default eligible changes to **bypass-uat** are:
  + Add/Remove line(s) in table enrichments & aliases
  + Property change (IP/Port/SCI/TCI etc.)
  + Plugin timer change

The PROD RFC that will be generated later will go through CAB validation.

###### Generating the PROD RFC

Once the UAT RFC is generated, as a CTB engineer, you will generate the PROD RFC, using the ticket number of the UAT RFC:



The generated PROD RFC will instruct the UL PROD team to:

1. Deploy the 1.2.1 (associated to git review 4253) in PROD.
2. Perform all the manual instructions associated to the release.

|  |
| --- |
| Phase 3:  CTB & FIX Teams Continue Working on the Implementation & Perform a Hot-Patch |

While the 1.2.1 version is being deployed/validated in UAT and then deployed in PROD, the CTB and FIX teams can already start working on new features:



The process is exactly the same as before. The easiest way is to create new **Feature Branches** starting from the Head, i.e. the latest commit on the **master** branch.

##### 

##### 

##### Performing a Hot-Patch

As the 1.2.1 started running in pre-production with some live end-to-end users, an error in an IP/Port configuration is discovered and one plugin cannot connect to the counterparty.

This is a critical issue for the PROD platform and it has to be fixed without waiting for the **Feature Branches** to get fully implemented, merged into **next-release**, etc. It requires a quick hot-patch, directly on the **master** branch.

As a CTB Engineer, you must:

1. Directly checkout the **master** branch.
2. Perform the IP/PORT change in it.
3. Commit it into your local **master** branch.
4. Push it to Gerrit.
5. Get it reviewed by the UL PROD team (see [Creating a Production-eligible Package](#_yoa022jzy8nl)).



You must then generate the UAT and PROD tickets, but this time, you can use the **bypass-uat** option on the UAT ticket, as you only touched the **prd-devop-pa\_ulbridge.properties**, which does not have any impact in UAT.

|  |  |
| --- | --- |
| **WARNING** | If, for any reason, going through the UL DEPLOY process is not possible (e.g. the fix needs to be done in 10 minutes), the person requesting the UL PROD team for the manipulation is **responsible for reporting this chang**e into the source control. |

## Part 2 - RTB Workflow

The platform is now live at our client’s. It has been handed over from the CTB to the RTB team. Its day-to-day life continues with some clients requests, additional implementations and bug fixes.

* [**Phase 4**](#_wzub76hkh0mm): The platform is now live, the RTB team performs bug fixes and CTB/FIX teams work on new features to implement.
* [**Phase 5**](#_2iimttitus6o): New bugs are fixed and the RTB team cherry picks a feature from the implementation branch.

|  |  |
| --- | --- |
| **WARNING** | From the moment the handover is done from CTB to RTB, the **CSM becomes responsible** for the **next-release** and **master** branches.  No one is supposed to commit to those two branches without the CSM’s consent. |

## 

|  |
| --- |
| Phase 4:  The Platform is Live, RTB Performs Bug Fixes as New Features are Implemented |

**Now, let’s say you are the CSM in charge of this client.**

In this phase of the scenario, you ask the Support Expert team to FIX two bugs on the platform, while the CTB team is working on implementing UL EDMA.



You then agree with the client that the next delivery in Production will only contain the bug fixes. Therefore you will merge the **bug-fixes** branch into **next-release**, as explained in [Phase 2.5](#_lmdos9u4vaxg).

Once the **next-release** branch isvalidated by the client, you can merge it into **master** and create a [Production eligible package](#_yoa022jzy8nl).

Depending on the content of your package, you will be able to get this package deployed during the week or during the weekend. In this scenario, it could be deployed intra-week, as the **bug-fixes** branches will only touch enrichments and CSV configuration (see [Hot/Cold Deployment](#_8la88kkqfjxy) in Appendix).

|  |
| --- |
| Phase 5:  New Bugs are Fixed, a Feature is Cherry-Picked from the Implementation Branch |

In this phase of the scenario, you (the CSM) have delivered 1.8.1 in Production, and the CTB team has continued working on a feature branch to implement UL EDMA and UL XMAN.

They have worked with the client, and with your consent, they have delivered their branch(es) in UAT for the client to test both features.

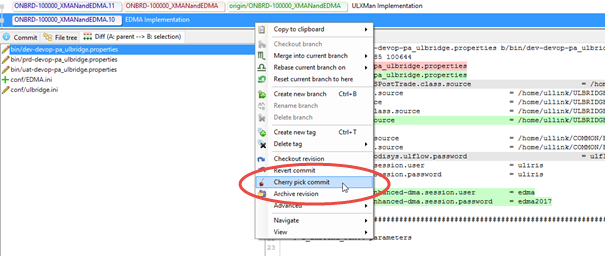
In the meantime, the client realises that a mapping was wrongly done on a CFB in Production. You therefore ask a Support Expert to fix the issue and to deliver this in the **bug-fixes** branch: as this is not a critical issue, there is no need for a hot-patch.

After discussing with the client, you agree that the **next-release** branch to go to Production should contain the bug fix and UL EDMA, but not UL XMAN (the best practice is for the client to validate all bug fixes in the bug fix branch to deliver them in PROD, but for features, it makes more sense to promote them at different times).

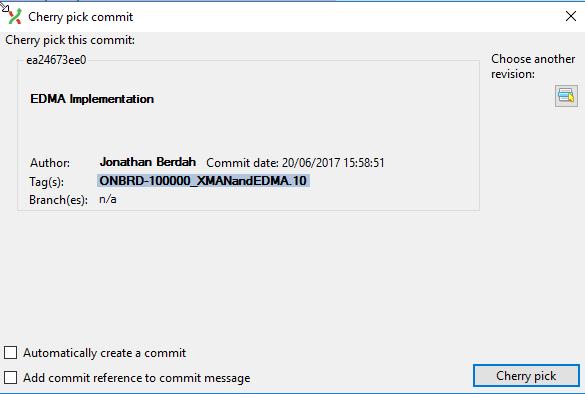
Therefore, you merge the **bug-fixes** branch into **next-release**, as in [Phase 4](#_wzub76hkh0mm), and then you **cherry-pick** the UL EDMA implementation commit:



To cherry-pick the UL EDMA implementation, right-click and select **Cherry pick commit**:



The **Cherry pick commit** popup opens:



Click **Cherry pick** to apply the modifications of the selected commit on top of the checked out branch (in this case, **next-release**).

# AUTOMATED TESTS

## 

## Add

# 

# APPENDIX

## Amend Commit

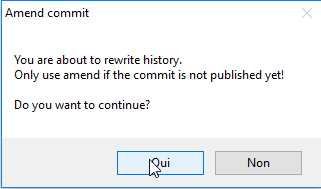
Once you have pushed a change to Gerrit, it will require a **+2** before it can be submitted into the repository. However, the following scenarios can occur:

* The review bot gives you a **-1**: the bot is automatically checking the diff upon submission, and will give you a -1 if your commit is not compliant with some rules (e.g. wrong tabs, spaces, usage of hardcoded IPs in the .ini, etc.).
* The reviewer (usually a UL PROD member when merging in **master**) gives you a **-1** or a **-2** because he or she detected something that is not Production-compliant.
* You realize that you did something wrong after pushing the change and you would like to amend your commit before asking someone to review.

In any of these 3 scenarios, you will decide to amend your commit (you could also abandon and resubmit a new commit but it would be more painful for both you and the reviewer).

To amend a pushed commit:

1. Your local repository must be checked out on the branch you committed on, and you can see the local commit that you pushed.
2. Modify your file system to fix what the review-bot/reviewer has rejected.
3. Open the **Commit** window and stage the modified files.
4. Check the **Amend Commit** checkbox, do not touch anything else.
5. Click **Commit**.
6. In the **Amend commit** popup, click **Yes**:



1. Push to Gerrit, on the right branch.   
   This time you will not get a popup with the Gerrit review link: you are just uploading a new “patchset” to your commit, it will be the same link for the reviewer. The review-bot will automatically review your commit once again.

You can repeat the Amend Commit procedure as many time as you want, until your commit is submitted.

## Parallel Commits

Sometimes, two implementation can occur in parallel on the same branch.

For example, CTB Engineer 1 and CTB Engineer 2 both check-out the **implementation** branch in parallel on their local computer, and perform a different implementation directly in it:



Let’s say that **CTB Engineer 1 pushes the commit to Gerrit first**. Two cases may occur:

**Case 1 - CTB Engineer 1’s Commit Gets a +2 and is Submitted to Gerrit**

When CTB Engineer 2 does a **fetch all** before pushing his/her change to Gerrit, the first commit (by CTB Engineer 1) will be fetched and CTB Engineer 2 will have to handle that situation on his/her local computer (integrate his/her change on top of what CTB Engineer 1 has done).

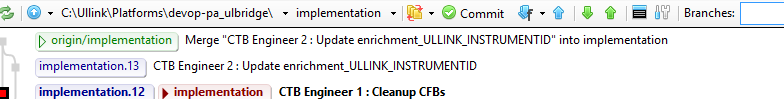
**Case 2 - CTB Engineer 1’s Commit is Not Submitted to Gerrit Yet**

When CTB Engineer 2 does a fetch all, CTB Engineer 1’s change will not be fetched. CTB Engineer 2 will then push his/her own change to Gerrit. In this situation, the following can happen:

**If the two changes are not conflicting**

When CTB Engineer 1 and CTB Engineer 2 submit their commits to Gerrit, both will work and each commit will have its own package built.

Gerrit will also automatically trigger a merge of these two changes, but this merge commit will not have a package available.



If there is a need to get a package including both changes, the trick would be to either commit a new change in the implementation branch and push it to Gerrit, or to manually trigger the build job in Jenkins, using the merge commit’s **commit hash**:



And in Jenkins:

Clicking **Build** will generate the package for the merge commit, push it to XL DEPLOY and also create the package tag in Git:

#### 

**If the two changes are conflicting**

If the two conflicting changes are waiting for validation in review

* The first change which to be submitted will go through the usual process: it will generate a package, etc.
* The second change will show a **Cannot merge** error message in Gerrit. The CTB Engineer will have to abandon that change in Gerrit, and work on his/her local Git repository to perform the merge and solve the conflicts.

## 

## 

## Deployment Instructions

Some changes that you might need to apply to the platform cannot be done through configuration and therefore will not be part of the package itself.

For example, uploading a new algorithm into UL AMS, creating a UL ODISYS user, deleting some instruments from UL ODISYS, etc.

These manual operations need to be executed by a UL PROD engineer, as well as does the deployment of the package (particularly when you are working on implementing a Feature, in Business As Usual mode on the RTB side, these manual operations can be directly done in an RFC without any package deployment).

In order to track these operations and to make sure they will be part of the generated RFC later on, you should write them down in the **manualOps/deployment-instructions.txt** file, which one of the files under source control (you must create it if it does not exist).

When merging multiple branches into **next-release**, the CTB Engineer/CSM might get different manual operations coming from the different branches: it is their responsibility to understand what needs to be done and to order the deployment instructions accordingly.

It is recommended to directly write the proper JMX command in this file exactly as the UL PROD engineer should execute it. As well, FIXatdl and other XML files with important static data that could be useful in the future should be committed in the source control (in **conf/staticdata/**): they will be released in the environment with the package, allowing the UL PROD engineer to not bother uploading them manually, and they will be available for future reference.

## 

## 

## Database Management

Let’s consider the following situation:



**Deploying the implementation.1 Package**

Package **1.10.1** is in UAT. During deployment of the  **implementation.1** package, the UAT database working with package **1.10.1** will be locally backed-up on the server (this will be called a “branch” database backup).

Now, when the RTB needs to perform a change in the Import, they will not want to include the UL ODISYS upgrade to V4 in the package that they will later deliver to Production. Therefore, they will work in the **next-release** branch (it could also be a hot-patch, or in a bug-fix branch) and build the **next-release.2** package.

**Deploying next-release.2 Package - Database Rollback**

When the RTB team decides to deploy the **next-release.2** package in UAT, by default, **UL ODISYS will no longer properly start**: the **next-release.2** package references a UL ODISYS V3, but the database on the UAT environment has been migrated to the UL ODISYS V4 model.

This is why, at deployment time, they will need to **rollback to a previous database**.They have two options:

* **Rollback to the “branch” database:**As explained above, when CTB deployed **implementation.1**, a backup of the database that was used with **1.10.1** has been locally created. Rollbacking to this version will work fine, but everything the client/CTB has done while **implementation.1** was running in UAT will not be available (static data, orders, trades, etc.).
* **Rollback to the “Prod” database:**   
  This rollback feature is mostly designed for Dress Rehearsals, but it can be useful if, for some reason, the “branch” database is not available (deleted by the Production team to make space, step skipped during deployment…). It will rollback to the daily backup of the Production database. For UL ODISYS, the “admin/admin” and “sales/sales” accounts will be automatically injected into the database and “TBLSERVERS” will be truncated (avoiding to wrongly connect to a Production Archive).

To perform a database rollback:

When deploying the platform with XL DEPLOY, before clicking **Execute**, click **Deployment Properties**:



The **Deployment Properties** popup opens. Select the database in the **Restore Database** menu:

## 

Selecting the **branch** database will look on the UAT disk for the latest backup of the branch of the package about to be deployed (in this case, **next-release**).

If it does not exist, it will fallback to the latest backup of the **master** branch (in our case, the backup of **1.10.1**).

During deployment, right before the “Are we ready to restart?” step, the following step will be added:

## 

Now that the **next-release.2** package is in UAT with a valid database, creating the **1.11.1** and deploying will not require a database rollback: it is basically the same package as **next-release.2** and the database will work perfectly fine.

**Deploying the implementation.3 Package**

When the CTB team decides to deploy **implementation.3**, they have 2 choices:

* **Rollback to the “branch” database:**This will rollback the backup of **implementation.1** (which was automatically created during the deployment of **next-release.2**). They will thus rollback on a UL ODISYS V4 database, with all the data that was created while **implementation.1** was running in UAT.
* **No rollback:**If they did not do anything specific on the database while **implementation.1** was running, they can deploy **implementation.3**. During the restart of UL ODISYS, the database in UAT will re-migrate to the UL ODISYS V4 model (but data created while **implementation.1** was running in UAT will be lost).

If they decide to rollback to the **implementation.1** database, the CTB will do the exact same procedure as above when deploying **implementation.3** through XL DEPLOY (selecting the **branch** database in the **Deployment properties** window before clicking **Execute**).

|  |  |
| --- | --- |
| **NOTE** | Database rollback can be done in DEV and in UAT, but is not available in Production. |

## Hot/Cold Deployment

**Branch package**

A deployment of a **branch package** in DEV or UAT will always be **cold**.

This means the application will always be stopped and restarted, allowing a full reload of the entire configuration, guaranteeing the fact that every change has been well applied.

It might feel like an overkill, but as the UL PROD team is the only team with admin rights and SSH access in UAT, it is very practical for the other teams to be allowed to deploy in UAT without requesting assistance from the UL PROD team.

**Master package**

**Master packages** are deployed in UAT and PROD by the UL PROD team.

In order to be able to deploy some packages intra-week (which means no restart of the platform), depending on which files are updated, the deployment can be done **hot**, i.e. it will just change the configuration files on disks, without stopping/starting the application.

XL DEPLOY will automatically choose between a cold and hot deployment depending on:

|  |  |
| --- | --- |
| **Product** | **Will force a cold deployment:** |
| **UL BRIDGE** | * Modification of **ulbridge.ini** * Modification of **version.parameters** |
| **UL ODISYS** | * Modification of **ul-odisys-server.ini** * Modification of **version.parameters** * Modification of any file in the **conf** folder that **is not** in:   + conf/enrichments/   + conf/extensions/ul-core/   + conf/modules/   + conf/staticdata/ |

The RFC Bot is aligned with that logic: when generating the RFCs for UAT and PROD, it will consider the changes done between the current deployed package and the one about to be deployed.

In case of **hot** deployment, it will instruct the plugins/enrichments/aliases/properties to reload **as well as it can**.

|  |  |
| --- | --- |
| **WARNING** | 1. The RFC Bot will not always be able to detect everything that needs to be reloaded: **it is your responsibility to review the generated RFC** and make sure that everything that needs to be reloaded is indicated and in the right order. 2. The dev/uat/prd.properties files are not considered as forcing a **cold** deployment: if you only modify these 3 files, your package will be **hot** deployed in UAT and PROD. This is very useful for changing ip/port/SCI/TCI/etc. properties, but be careful when upgrading a plugin/module on the UL BRIDGE side, as this has to be done in **cold** mode.  The workaround for now is to adapt your RFC to inform the UL PROD team about this:  this will anyway be a week-end change and will be deployed on a stopped platform, but the deployment will not automatically restart the platform at the end (hot deployment just changes the configuration files on the disk).   In the future, we plan to move the link to the **.jar** files out of the dev/uat/prd.properties file and this will not be the case anymore. |

## 

## 

## 

## 

# FOR MORE INFORMATION…

## Contact

Jonathan Berdah - [jonathan.berdah@ullink.com](mailto:jonathan.berdah@ullink.com) (any questions, process, tools, etc.)

Jessé Liben - [jesse.liben@ullink.com](mailto:jesse.liben@ullink.com) (UL CLONE-related issues & questions)

François Hervieux - [francois.hervieux@ullink.com](mailto:francois.hervieux@ullink.com) (Bot-related issues & questions)

Guillaume Laurent - [guillaume.laurent@ullink.com](mailto:francois.hervieux@ullink.com) (XL DEPLOY-related issues & questions)

Or join the [**#ul-deploy**](https://gaia.slack.com/messages/C4EHBM3K2/details/)Slack channel.

## Other Documentation

### Git Documentation

<https://git-scm.com/doc>

### Gerrit Documentation

<https://gerrit-documentation.storage.googleapis.com/Documentation/2.14.1/index.html>

### XL DEPLOY Documentation

<https://docs.xebialabs.com/xl-deploy/>

|  |  |  |  |
| --- | --- | --- | --- |
| **ABOUT ULLINK**  Ullink is a global provider of market leading multi-asset trading technology and infrastructure for buy-side and sell-side market participants, including NYFIX, one of the industry’s largest FIX based trading communities. Trusted by over 150 of the world’s top-tier banks and brokers, the company provides consistent, reliable access to the most current and innovative trading solutions available. Our multi-asset solutions allow you to connect to your desired markets, trade when and where you want, while being able to comply with global regulation. From our 10 offices covering all the major global financial centers, our team will advise on the best approach to address your challenges and deploy our technology. | | | |
| **AMERICAS** | **EUROPE** | **APAC** | |
| New York +1 212 991 0816  São Paulo +55 11 3171 2409 | Paris +33 1 49 95 30 00  London +44 20 7743 7200  Cluj +40 364 803 200 | Hong Kong +852 2521 5400  Tokyo +81 3 3664 4160  Sydney +61 2 9247 3633 | Singapore +65 6221 7607  Manila +63 2459 4801 |