Configuration Scripts

Overview

Mission control embraces the configuration-as-code approach and uses scripts to configure the services that it manages. Scripts are reusable pieces of code which can be applied to one or more services at a time to perform a variety of actions. These can range from simple configurations, such as setting a new caching policy for a set of remote repositories in an Artifactory service, to more complex ones like creating a combination of local and remote repositories in a master-slave topology, or even to set up watches in a managed Xray service.

Here is a simple example of a configuration script that creates a local repository that will be a Docker registry in a managed Artifactory service called "Art1":

```
artifactory('Art1'){
   localRepository("docker-local") {
     packageType "docker"
     description "My local Docker registry"
   }
}
```

Using configuration scripts presents several benefits for the management of services under Mission Control

- **Automation:** Configuration scripts improve your efficiency by enabling you to automate your service configuration tasks, preventing the need to perform repetitive and error prone manual configuration, especially when managing multiple services.
- **Reliability:** Configuration scripts improve the reliability of your configuration tasks by letting you reuse the same configuration on multiple services that may also be running on different runtime environments (e.g. development, staging, production).
- **Standardization:** Configuration scripts can be used to enforce standards when configuring things such as repository names, include/exclude patterns, caching policies and more.

Scripting in version 2.x

In version 2.0, scripting in Mission Control underwent significant changes. Any scripts written for Mission Control v1.x will not work and need to be migrated. For details, please refer to Migrating Scripts from Version 1.x to Version 2.x.

Page Contents

Read More

- Configuration DSL
- Git Integration

Working with Scripts

Scripts are written using the Groovy programming language and are managed under version control in a Git repository. This allows you to edit scripts directly in your Git provider's editor, or using any other editor you prefer to work with. Once your scripts are committed to your Git repository, Mission Control accesses them using the configuration you specify under Git Integration and is automatically synchronized with any additions or deletions you make from the Git repository.

Create vs. Update

Mission Control scripts are written in the same way whether they create new entities or just update them. If the relevant entity (service or repository) already exists on the target service, then it will just be updated with the parameters specified in the script. If the entity does not exist, it will be created with the parameters specified. Any optional parameters not specified will take default values.

For example, consider this simple script :

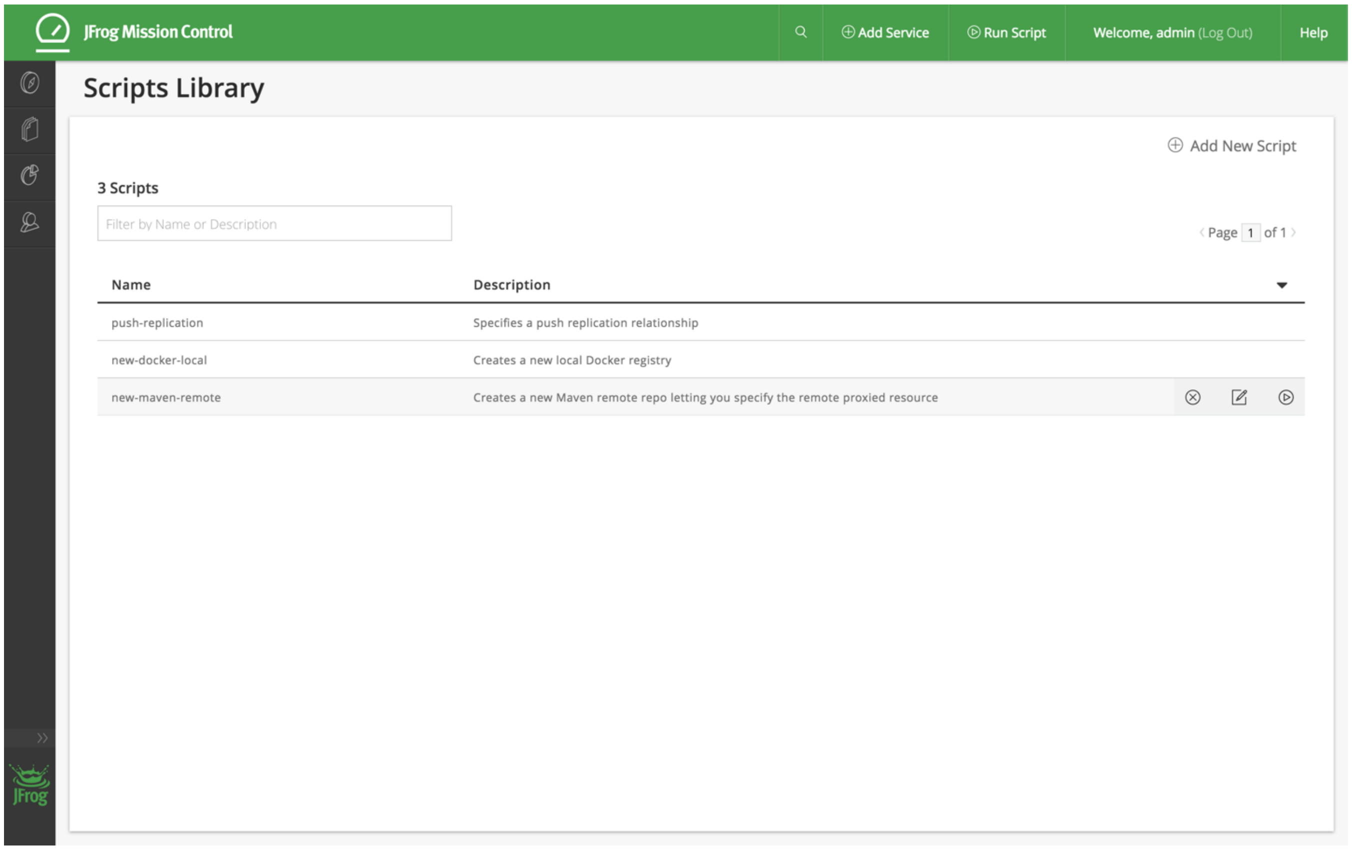
```
artifactory('Art1'){
   localRepository("maven-local-dev") {
     packageType "maven"
     description "This is my Maven repository for development"
   }
}
```

If the "Art1" Artifactory service already has a Maven repository called "maven-local-dev", its description will be updated to the value provided. If the repository does not exist, it will be created with the description provided and all other parameters set to their default values.

Note that when doing a dry run on a script, Mission Control will show the changes that the script will implement in the case of an update.

Script Library

Scripts are managed in the Script Library which can be accessed from the **Admin** module under **Scripts | Script Lib.** Hovering over an item in the library displays icons that let you **delete**, **edit** or **run** the script. Scripts are maintained in a MongoDB database which can only be accessed by Mission Control. If you have configured Git Integration, all scripts will be synchronized with the Git repository specified.



Scripts Library

3 Scripts

Name	Description
push-replication	Specifies a push replication relationship
new-docker-local	Creates a new local Docker registry
new-maven-remote	Creates a new Maven remote repo letting you specify the remote proxied resource

Running a Script

Running a script involves the following steps:

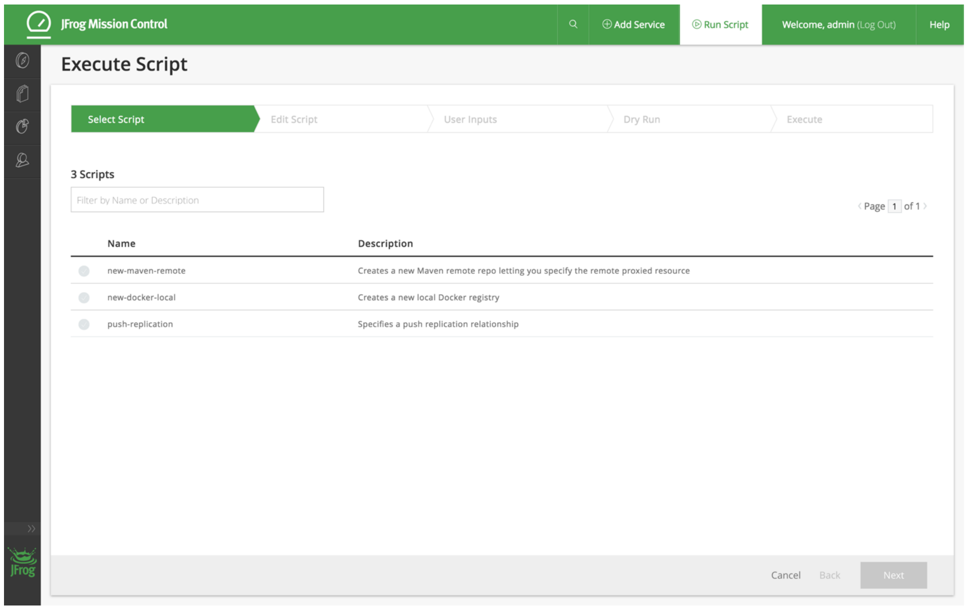
- **Selection** - select the script you want to run out of the list
- **Editing** - make any edits needed to the script
- **Providing input** - provide any input required by the script
- **Dry run** - do a dry run of the script to ensure there are no errors
- **Execute** - execute the script

These main steps are described in more detail in the sections below.

Selecting a Script to Run

There are two ways to select a script to run:

1. Selecting **Run Script** from the Mission Control top ribbon. This will present the list of scripts available in your script library



Execute Script

3 Scripts

Name	Description
new-maven-remote	Creates a new Maven remote repo letting you specify the remote proxied resource
new-docker-local	Creates a new local Docker registry
push-replication	Specifies a push replication relationship

   Select the script you want to run and click "Next".
2. Hovering over the script in the Script Library and clicking the **Run** icon.

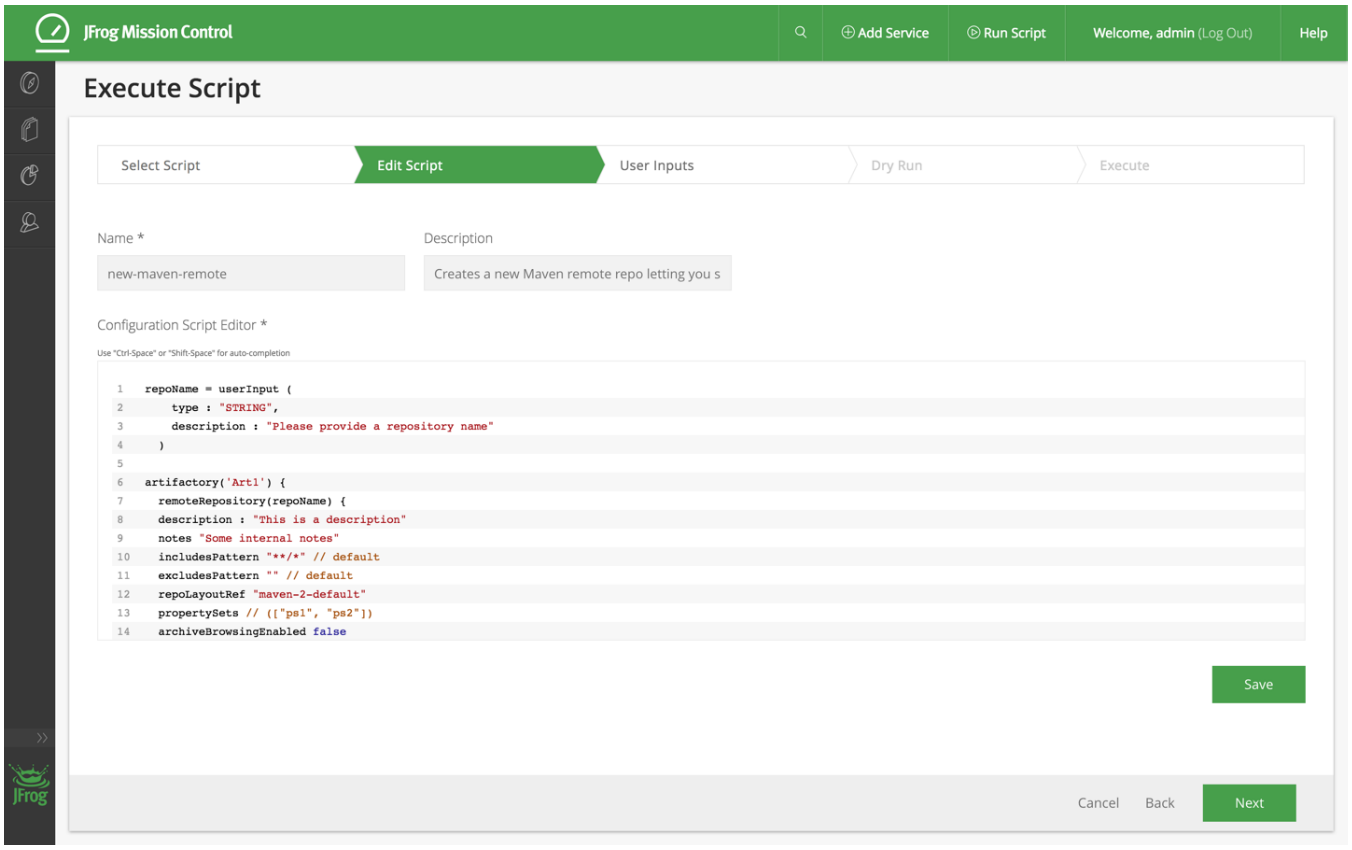
Both of these actions will take you to the next step of editing the script.

Editing a Script

The **Edit Script** tab lets you make any changes necessary to the script to meet the specific needs of the current run.

Tweak, don't make big changes

This feature was designed to let you make minor changes that you may need to make to accommodate slightly different scenarios to which you would apply a script. To make significant changes to a script, we recommend modifying the script using an external editor and committing it to your Git repository.



Name *

new-maven-remote

Description

Creates a new Maven remote repo letting you s

Configuration Script Editor *

Use "Ctrl-Space" or "Shift-Space" for auto-completion

```
repoName = userInput (
    type : "STRING",
    description : "Please provide a repository name"
  )

artifactory('Art1') {
  remoteRepository(repoName) {
  description : "This is a description"
  notes "Some internal notes"
  includesPattern "**/*" // default
  excludesPattern "" // default
  repoLayoutRef "maven-2-default"
  propertySets // (["ps1", "ps2"])
  archiveBrowsingEnabled false
```

Note that you cannot modify the name or description of an existing script.

Click "Save" to save your changes.

Saving changes commits them to Git

Note that if you have a Git repository defined for your scripts, saving your changes will commit them to Git.

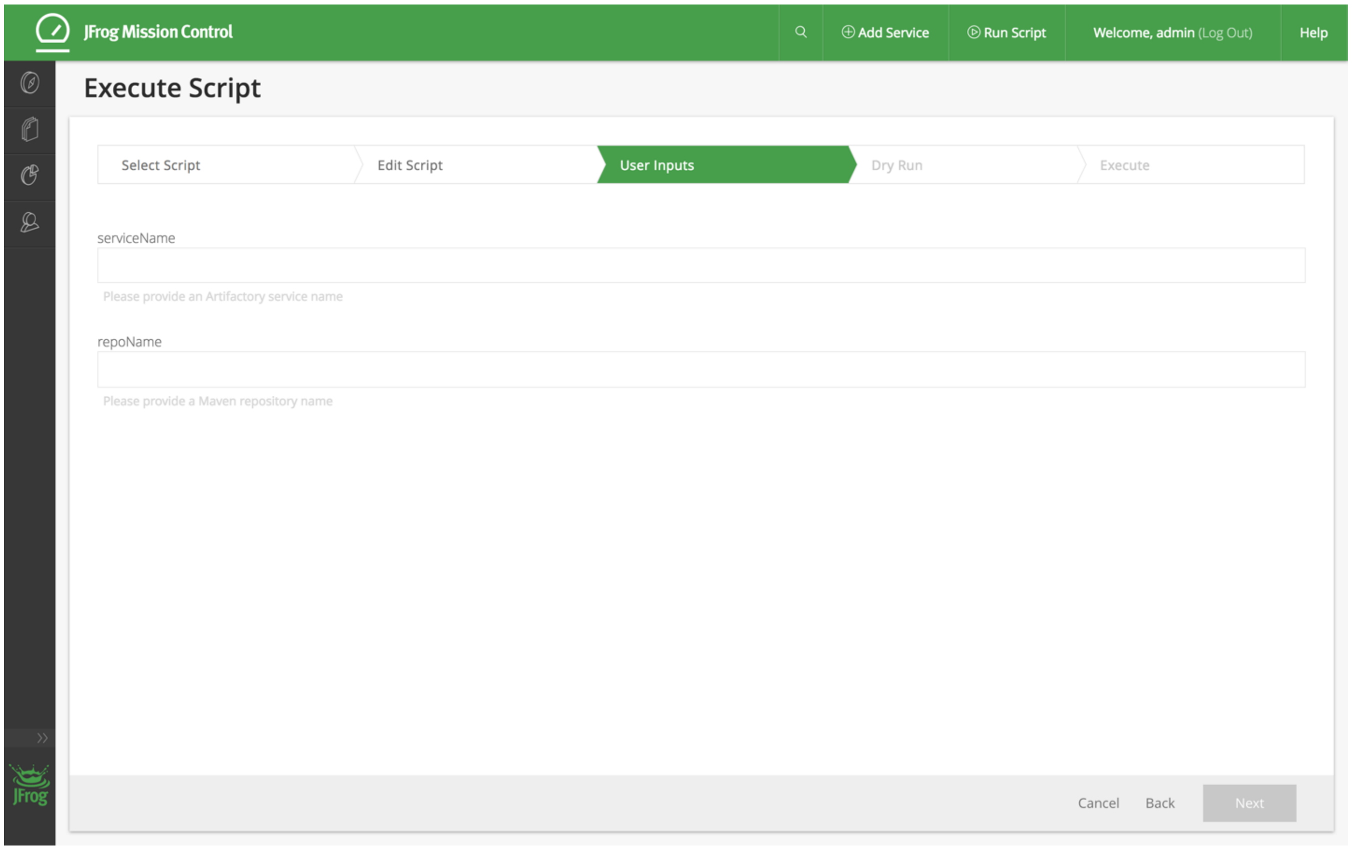
Syntax Errors

If there are any syntax errors in your script, Mission Control will display an alert. You need to fix the error in order to proceed.

Click "Next" to move on to the next step of adding user input.

Entering User Input

This is the step in which you provide any user input required by the script you are running.



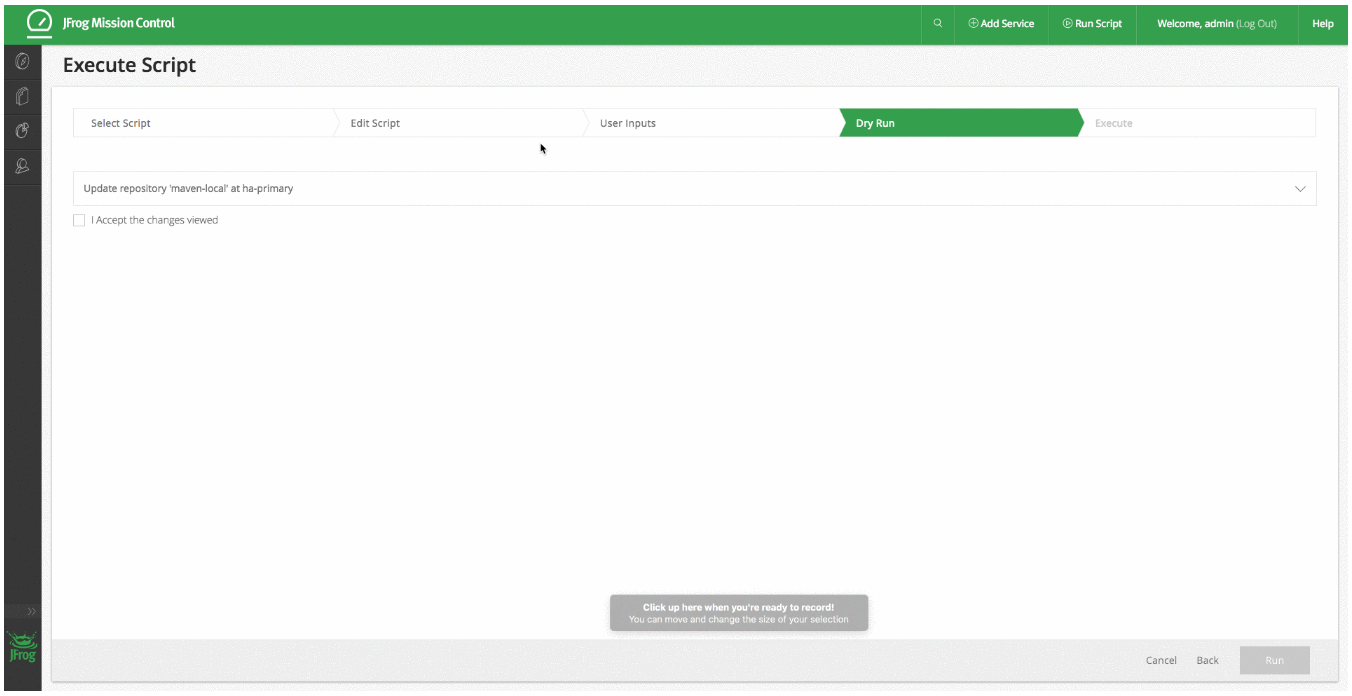
Enter the user input required and click "Next" to move on to the next step of testing your script in a dry run.

If there are invalid instructions in your script, Mission Control will display an error. You need to fix the error in order to proceed.

For example, if you have written a script that updates an Artifactory service called "Art1", but there is no such service recognized by Mission Control, then this is an error.

Doing a Dry Run

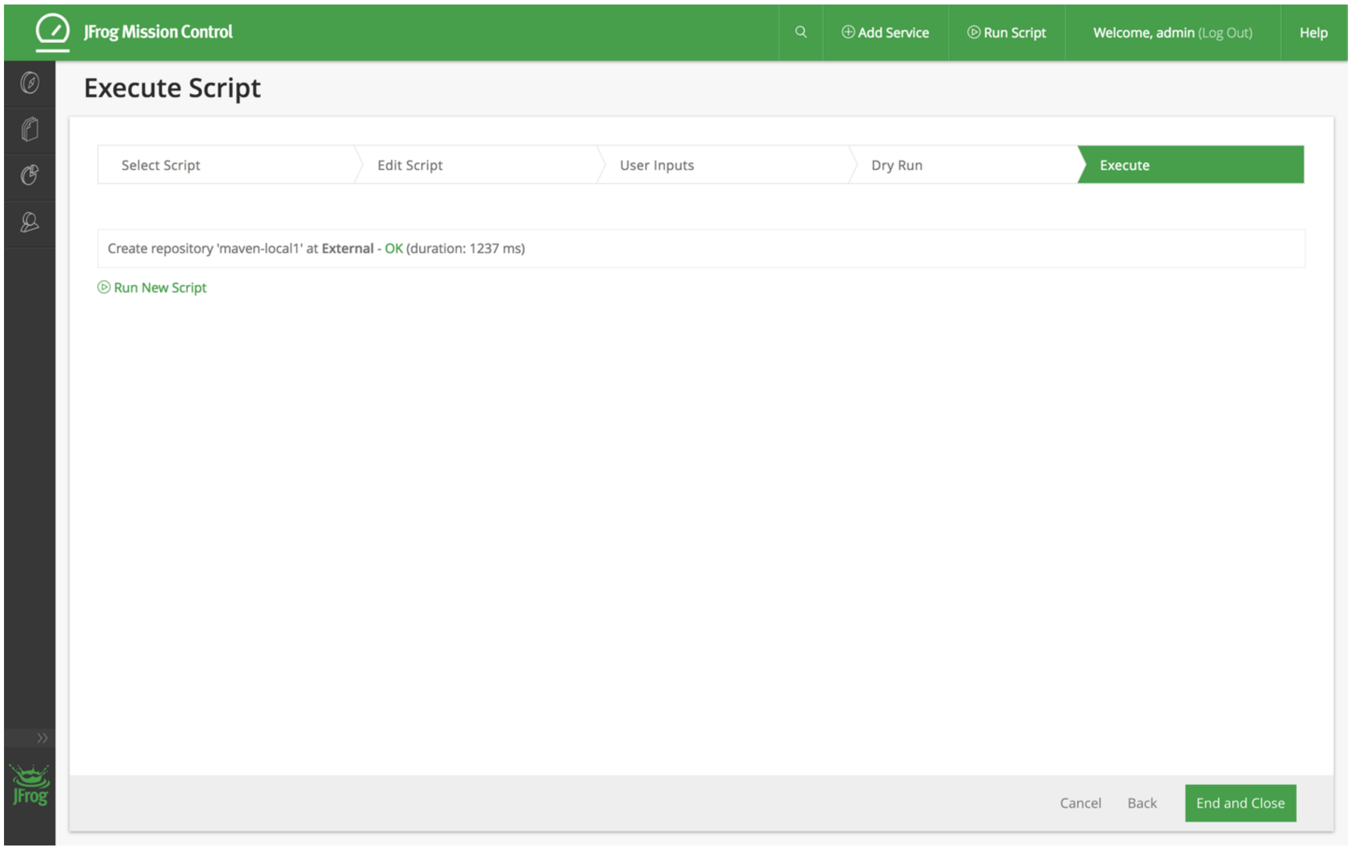
The dry run does not execute the script, but let's you know what changes will be implemented on the selected services and repositories when you do. To see the changes, click the the summary line.


JFrog Mission Control
Add Service
Run Script
Welcome, admin (Log Out)
Help
Execute Script
Select Script
Edit Script
User Inputs
Dry Run
Execute
Update repository 'maven-local' at ha-primary
I Accept the changes viewed
Click up here when you're ready to record!
You can move and change the size of your selection
Cancel
Back
Run

To do a dry run, accept the changes your script will make and click "Run".

Executing a Script

If the dry run is successful, you can click **Run New Script** to actually execute the code in the script. Take care, and note that this time, any changes programmed into your script will actually be executed on the corresponding services and repositories.



Create repository 'maven-local1' at **External** - OK (duration: 1237 ms)

Upon successful execution, Mission Control will again display the list of scripts available for you to run.

At any step along the way, you can stop the process of running a script by clicking "Cancel", or by clicking "End and Close" in the Execute step.

Creating and Editing Scripts

As described above, scripts are written in Groovy.

There are two ways to create and edit scripts:

- Using an external editor
- Using the Script Editor

> ✓ **Use and external editor**
>
> As a best practice, we recommend creating scripts outside of Mission Control using your preferred editor and then committing them to the Git repository configured in the Version Control page.

Using the Script Editor

To create a script from within Mission Control, select **Add New Script** in the Scripts Library. To edit the script select the "Edit" icon while hovering over the script's entry in the Script Library.

> ✓ When you have a Git repository configured in the Version Control page, Mission Control will commit any new scripts you create in the script editor to the Git repository and commit new versions when you edit the scripts.

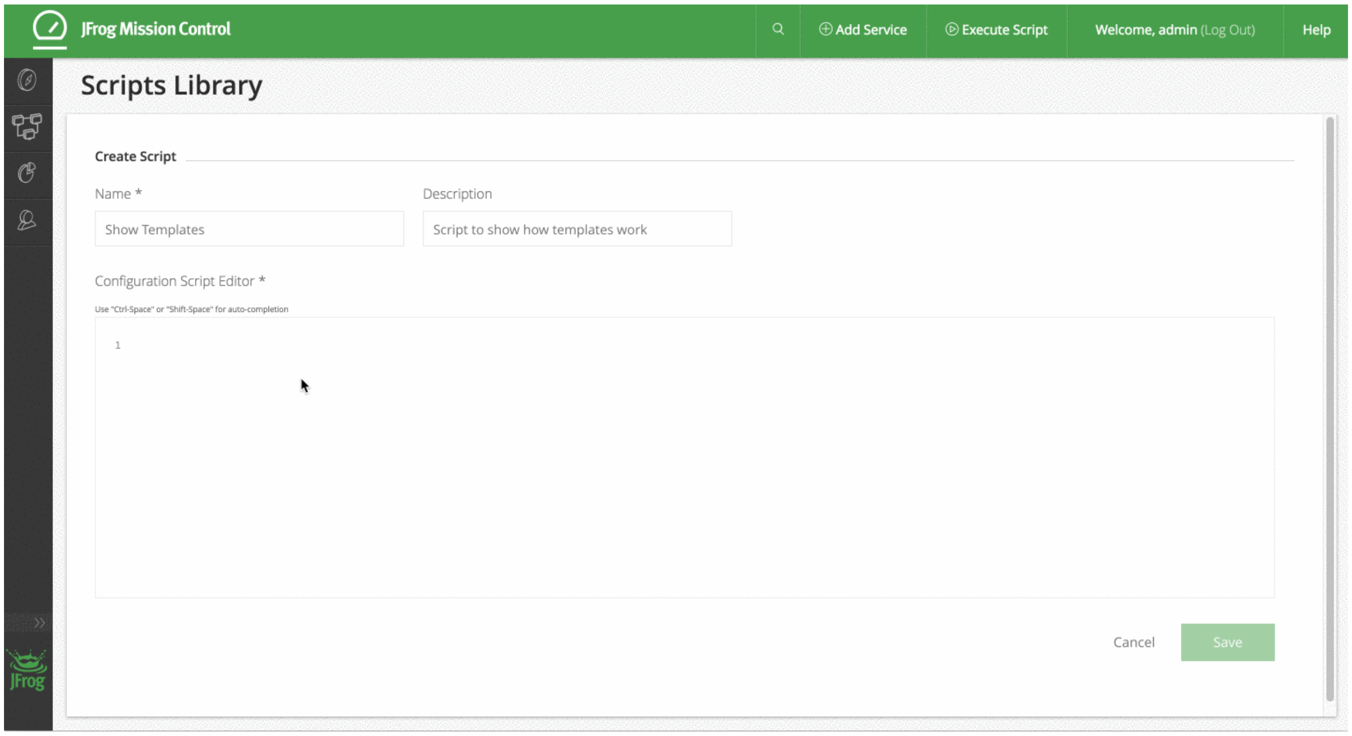
Once in the script editor you can create and/or edit the script as needed. When done, click "Save" to save the script. If a Git repository is configured, Mission Control will commit your changes.

Block Templates

As a convenience, Mission Control offers configuration blocks as built-in templates that you can use when creating or editing scripts. The templates available are exposed by typing the first few letters and then **ctrl + space** or **shift + space** to show the auto-complete options. When you select a template from the list, Mission Control will insert it into your script with all the parameters of the selected configuration block.

Example 1

The example below shows how to insert an Artifactory service and Local repository template:



Scripts Library

Create Script

Name *: Show Templates

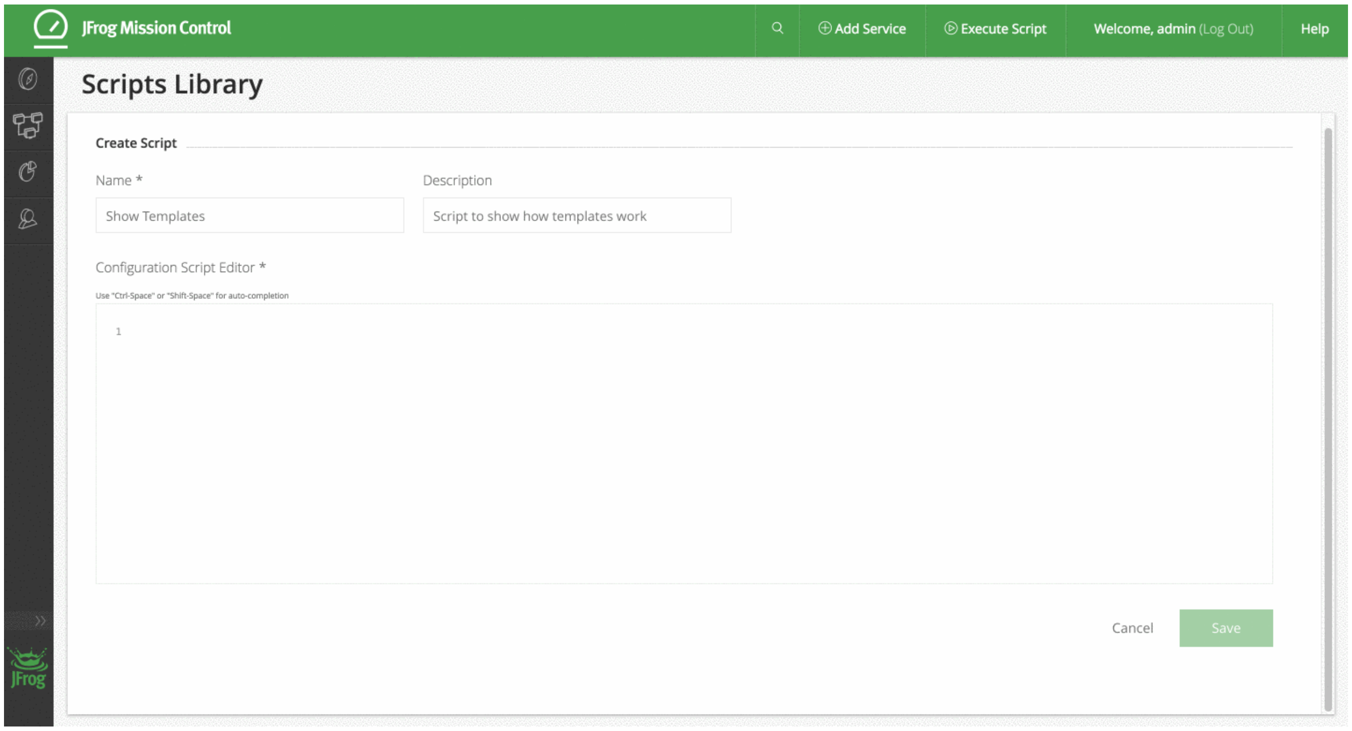
Description: Script to show how templates work

Configuration Script Editor *

Use "Ctrl-Space" or "Shift-Space" for auto-completion

Example 2

The example below shows how to add an Xray service template:



Using an External Editor

You can create or edit scripts using any external editor. Just make sure to commit any changes to the Git repository configured in the Version Control p age.

Script Elements

Scripts are constructed from Service Closures, Configuration Blocks and Properties. Scripts are very flexible, and can contain any number of service closures, each of which may contain any number of configuration blocks and properties.

For a complete list and full specification of service closures, configuration blocks and properties available, please refer to **Configuration DSL**.

Service Closures

Service closures define the service (Artifactory or Xray) that the script acts on. Artifatory or Xray services can only be created or modified through configuration blocks which must be enclosed in service closures.

Configuration Blocks

Configuration blocks define the parameters or changes that should be implemented on the services specified in their enclosing service closures.

Configuration blocks must be in service closures

A configuration block must be placed inside a service closure that specifies the service on which it should be applied.

Properties

Properties are used to configure the relevant parameters of a configuration block.

Example

The simple example script below shows:

- An Artifactory closure that includes a configuration block that creates a generic local repository called "generic-local" on an Artifactory service called "Art1". The configuration block includes a property that specifies the package type for the repository
- An Xray closure that connects an Xray service called "X1" to the Artifactory service "Art1" for indexing.

Example script

```
artifactory('Art1') {
  localRepository('generic-local') {
     packageType "generic"
  }
}

xray('X1') {
  binaryManager('Art1')
}
```

User Input

Static configuration scripts can be useful in some cases. For example, when you would like to create a property set with pre-defined properties. In other cases you need the script to be more dynamic. For example, when creating a new repository you might want to provide the repository name only when the script is applied.

To create more dynamic configuration scripts, the Mission Control configuration DSL lets you ask for user input when the script is applied.

There are two ways of declaring that you would like to get user input:

1. Asking for user input for a specific property

   ```
   localRepository("my-repository") {
     description userInput (
       type : "STRING",
       value : "This is a generic description",
       description : "Please provide a description"
     )
   }
   ```

   In this example, we ask the user to provide a value for the `description` property of a repository.

2. Assigning the user input to a variable and using the variable

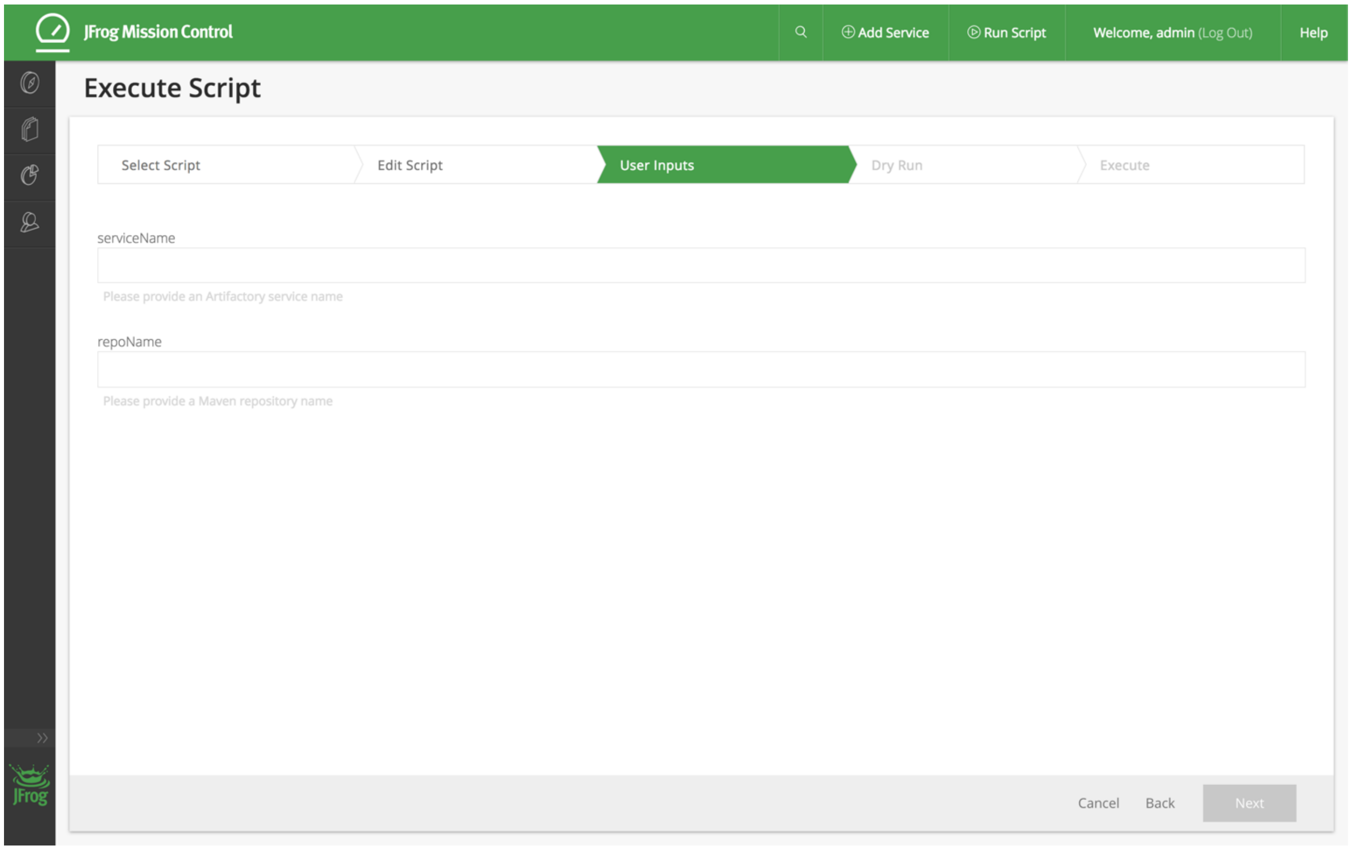
   ```
   name = userInput (
       type : "STRING",
       value : "This is a generic description",
       description : "Please provide a repository name"
     )

   localRepository(name) {
     ...
   }
   ```

   In this example, the value of the `name` variable is used in a Groovy string to create the repository name.

   Don't use "def"

   Take care not to use "def" when declaring user input for a script (e.g. `def name = userInput...`). When using def, the script will not work correctly as it will refer to the user input object rather than the dynamic value entered by the user

When a configuration script with user input is applied, Mission Control will generate a form that prompts you for all user input fields defined. The user applying the script will need to provide the input fields in order to proceed with a dry run and execution of the script.



Input Types and Properties

When requesting user input in a script, you need to specify the following parameters:

type	This can take one of the following values: STRING - the input is a simple string BOOLEAN - the input is a simple boolean INTEGER - the input is a simple integer SERVICE - the input is one of the services managed by Mission Control. The user input screen will display a list of services for the user to select from. ARTIFACTORY - the input is an Artifactory service managed by Mission Control. The user input screen will display a list of Artifactory services for the user to select from. REPOSITORY - the input is a repository in an Artifactory service managed by Mission Control. The user input screen will display a list of Artifactory services for the user to select from. XRAY - the input is an Xray service managed by Mission Control. The user input screen will display a list of Xray services for the user to select from. PACKAGE_TYPE - The input is one of the package types supported by Artifactory (e.g. "docker", "npm" "debian"). For a full list of supported package types, please refer to Repository Configuration JSON in the Artifactory User Guide.
value (Optional)	A default value. If a default value is not specified, the variable becomes mandatory and the user must provide the input string.
multivalued (Optional)	When true, mission control will allow the user to provide more than one value. The following types can take multiple values: SERVICE, ARTIFACTORY, XRAY, REPOSITORY
description (Optional)	A description to be displayed in the web UI

According to the input type requested, once entered, the script has access to different properties related to the input value as described in the table below:

Input Type	Property Name	Property Type	Description
SERVICE, ARTIFACTORY or XRAY	name	String	Service name in mission control
	url	String	Service URL
	type	EntityType (enum)	Service type (ARTIFACTORY, XRAY)
	description	String	Service description
	credentials	Credentials	Service credentials, ex: credentials.userName, credentials.password
REPOSITORY	url	String	Service URL
	repository	LocalRepositoryImpl, RemoteRepositoryImpl, VirtualRepositoryImpl	Repository properties as described in Repository Configuration JSON

Example

The following script requests a target Artifactory service as user input to create a replication relationship with another Artifactory service called "LocalK".

Once the target instance has been provided, the script uses its **url** property to specify its **maven-local** repository as the replication target.

```
targetArtifactory = userInput (
    name : "Target Artifactory",
    type : "ARTIFACTORY",
    description : "please provide the artifactory instance you want to replicate to"
)

artifactory('Local') {
  localRepository("maven-local") {
    replication {
      url "${targetArtifactory.url}/maven-local"
      username targetArtifactory.credentials.userName
      password targetArtifactory.credentials.password
      cronExp "0 0/9 14 * * ?"
      socketTimeoutMillis 15000
    }
  }
}
```

Global Variables

In addition to the objects and properties available to scripts as a result of user input, scripts also have access to a global variable called **services**.

The **services** variable is a map of all services being managed by Mission Control and provides access to the service object using its **name** property as the key.

For example, if Mission Control is managing an Artifactory service called "Art1" and an Xray service called "X1", a script can access these service objects using the following lines:

```
def serviceArtifactory = services['Art1']
def serviceXray - services['X1']
```

Once the respective service objects are acquired, the script can then reference the different properties such as name, url, description etc. as described in the table above.

Running Scripts via REST API

The REST API for interacting with and running scripts has changed in Mission Control 2.0. For details, please refer to SCRIPTS in the Mission Control REST API documentation.

Migrating Scripts from Version 1.x to Version 2.x

Following an upgrade of Mission Control from version 1.x to version 2.x, all scripts created with version 1.x should be available in the Script Library of version 2.x.

In version 2.0 JFrog Mission Control scripting functionality underwent several changes. The most significant change is in how you select which services to apply the script to.

In version 1.x, you could create scripts that operated on Artifactory instances and repositories, but would only have to select those instances and repositories when actually running the script.

From version 2.0 any operations on services (whether Artifactory or Xray) must be enclosed in a service closure (as described above) that specifies the service on which the script should be applied. While the specific service may also be entered with user input, the enclosing closure must be present in the script. As a result, you need to migrate all scripts written for version 1.x and **enclose the content of the script in a service closure**. The following example shows how a simple script written for version 1.x would be migrated to be compatible with version 2.x

Example

Create a local Docker registry called "docker-local" in an Artiafctory service.
In version 1.x, the Artifactory service would be selected during the flow of running the script.
In version 2.x we add an `artifactory` closure

Script in version 1.x	Script migrated for version 2.x

`localRepository("docker-local") {`
`  packageType "docker"`
`  description "My local Docker registry"`
`}`
	Option 1: Specify a specific Artifactory service called "Art1"
	
	`artifactory('Art1'){`
	`   localRepository("docker-local") {`
	`     packageType "docker"`
	`     description "My local Docker registry"`
	`   }`
	`}`
	
	Option 2: Let the user select the Artifactory service with user input:
	
	`whichArtifactory = userInput (`
	`    type : "ARTIFACTORY",`
	`    name : "Which Artifactory",`
	`    description : "Please specify the Artifactory service on which to create the repository"`
	`  )`
	
	`artifactory(whichArtifactory.name){`
	`   localRepository("docker-local") {`
	`     packageType "docker"`
	`     description "My local Docker registry"`
	`   }`
	`}`

Best Practices

Here are some best practices that we recommend you follow when migrating your scripts to Mission Control 2.0 and above:

Aggregating Scripts

In version 1.x, Mission Control scripts could only perform one action, and were limited to acting either on an Artifactory instance, or on a repository. From version 2.0 scripting is much more flexible and scripts can be written to perform any number of actions. We recommend taking a series of small, single-action scripts and aggregating them into a larger script that preforms several functions.

Dry Run

To make sure you have migrated your scripts correctly, we recommend doing a dry run and verifying the changes that Mission Control would make if you were to actually run the script.

Best Practices

To learn more about configuration scripts by example, check out JFrog's public GitHub repository of scripts for Mission Control.

These scripts provide best practices for writing scripts related to:

- Creating a single generic repository
- Onboarding a team of users
- Setting up replication relationships including implementation of a Star Topology and Mesh Topology

We recommend watching this repository for updates with more script examples.