The small RNA Workbench:

A Developers Guide

Project setup 2

Adding new tools (post Version 4) 3

Following the Model View Controller (MVC) paradigm 3

Model: 3

View: 3

Controller: 3

Workflow Modules 3

Preconfigured Workflows 4

Releasing the program 5

# Project setup

The workbench is a maven project. All dependencies should be handled with the POM file as should all build instructions.

SQL interactions are handled using the Hibernate system.

Data source configurations and any communication with REST services should be done using Spring which is already included in the project.

Each new tool should be placed in the package following the convention:

uk.ac.uea.cmp.srnaworkbench.tools.newtool

Each new tool requires a package for JavaFX code (listed as packageNameFX) and workflow code (packagenameWF)

# Adding new tools (post Version 4)

## Following the Model View Controller (MVC) paradigm

Model: The sRNA Workbench workflow module code

View: Composed of one or more HTML Files, one FXML file and one or more Javascript programs (plus any other web scripts you may wish to use). See any example HTML file for a complete run down of which js files should be included and how to create the workbench style menu system.

Controller: A special class in the workbench registered as the controller for the FXML file (do not do this in the FXML markup but instead do it programmatically, see other controller classes for examples). In addition to implementing the Initializable interface, ControlledScreen should also be implemented.

An internal class to act as a bridge between the view and the controller should be added. This class should extend the JavascriptBridge class and take a ScreensController controller as a parameter. This parameter should be passed to the superclass via the constructor.

The bridge object should be a class field to ensure GC processes do not remove it when it is seemingly unused (as it will be used by the javascript engine Java may not be aware of it). It can be instantiated and set as a member in the JS engine using the following example code:

mainWebEngine.getLoadWorker().stateProperty().addListener((ObservableValue<? **extends** State> ov, State oldState, State newState) ->

{

myJSBridge = **new** JavascriptReceiver(**this**.myController);

**if** (newState == State.SUCCEEDED)

{

JSObject window = (JSObject) mainWebEngine.executeScript("window");

window.setMember("app", myJSBridge);

}

});

## Workflow Modules

Workflow modules can be chained together to form a workflow. Each module consists of two classes that must extend the correct interfaces and abstract classes. A complete new module’s functional code will be one module and one module runner (graphical element is explained above). A complete example of a default (clean) new tool can be found in the docs directory.

Modules must be connected together to form a workflow. Workflows always start with either the Database or FileManager module. One builds an on disk or in memory database to store workbench data at run time. One simply serves as the input module for files. Both have the same interface.

More information on the workflow modules currently available in the workbench can be found on the workbench website including tutorials on their use.

### Preconfigured Workflows

Modules are chained together by adding them to the workflow manager:

First create the entry point to the workflow (either FileManager or Database) and their respective runner code and add them to the workflow manager class for example:

**Database Mode:**

DatabaseWorkflowRunner DB\_runner = new DatabaseWorkflowRunner();

WorkflowManager.getInstance().addModule(DB\_runner);

*//set size of file hierarchy web view*

DatabaseWorkflowModule.getInstance().setFrameSize(visualBounds);

DatabaseWorkflowModule.getInstance().setPos(60, (float) visualBounds.getMaxY() / 4.0f);

**Or using FileManager (no database)**

WorkflowManager.getInstance().setFirstModuleTitle("FileManager");

*//create the file manager module and add it to the workflow*

FileManagerWorkflowModule fm\_module = new FileManagerWorkflowModule("FileManager", visualBounds);

fm\_module.setPos(60, (float) visualBounds.getMaxY() / 4);

FileManagerWorkflowModuleRunner fm\_runner = new FileManagerWorkflowModuleRunner(fm\_module);

WorkflowManager.getInstance().addModule(fm\_runner);

Then add any further modules you require and connect them together.

**Example of connecting FileManager to Filter tool:**

//create the filter module and add it to the workflow

Filter2WorkflowModule filter\_module = new Filter2WorkflowModule("Filter", visualBounds);

filter\_module.setPos(260, (float) visualBounds.getMaxY() / 4);

Filter2WorkflowModuleRunner filter\_runner = new Filter2WorkflowModuleRunner(filter\_module);

WorkflowManager.getInstance().addModule(filter\_runner);

*//connect the FileManager to the Fitler tool*

WorkflowManager.getInstance().connectModule2Module("FileManager", "Filter");

This will mean that when the workflow is triggered the connected modules must complete before the module it is connected to will begin. Each completion of a module triggers an “update” event in which the next module on the list is queried to see if all its connections are also complete.

Modules can also be connected with SQL queries given as the data transfer. Examples of this can be found in the miRCat preconfigured workflows and the miRPare workflows.

When the correct interfaces are implemented the user will need to (among other methods that are related to graphical input) implement a process method. All functional code should go in here, as the method terminates the update method will be triggered in the workflow manager.

# Releasing the program

When ready to do a release, package up the software however you are familiar with. You can release a package with just jars or if you are familiar with creation of individual OS packages (.app, .exe etc) then those are fine too.