ECSE 321 - Intro to Software Engineering
Release Pipeline Plan - Deliverable 4

Harley Wiltzer
Camilo Garcia La Rotta
Jake Shnaidman
Robert Attard
Matthew Lesko

April 2, 2017

Contents

I Integration **2**

1 Description . . . 3
2 Tools . . . 3
3 Design . . . 3
4 Rationale . . . 3

II Build **4**

5 Description . . . 5
6 Tools . . . 5
7 Design . . . 5
7.1 Concerning the unification of the application platforms . . . 6
7.2 The calm before the storm: an overview of the intended file structure . . . 6
7.3 Build targets by platform . . . 7
8 Rationale . . . 9

III Deployment **10**

9 Description . . . 11
10 Tools . . . 11
11 Design . . . 11
12 Rationale . . . 14

Part I

Integration

1 Description

2 Tools

3 Design

4 Rationale

Part II

Build

5 Description

The build pipeline is responsible for the successful and efficient building of the source code and automated testing. As the system evolves, the processes required to carry out certain tasks like compiling the code, creating executables, and running tests will evolve as well, and will ultimately become more complex as time fades away. For this reason, it is important to design a robust automated build system that provides the developers with the power to do any of the aforementioned tasks by running simple build commands.

Of course, such power does not come from nothing; in fact, the developers must actively maintain this build system such that it is always kept up to date with the rest of the system. Fortunately, certain tools exist for creating such build systems that will significantly aid in the process. These tools will be discussed in the Tools section.

6 Tools

Although designing a Monad in Haskell for automating build procedures sounds very interesting, such a task would be a rather horrendous decision. Given the plethora of tools available that can already be used to aid in the development of build systems, it is much preferred to use tools that have already been tested and have already been shown to be efficient rather than developing one anew.

Firstly, designing and implementing build scripts, the Apache Ant tool will be used. A major convenience of this tool is that it has support for both Java and PHP languages, which cover 100% of the languages used to implement the TAMAS system. Ant provides the ability to design multiple *build targets*, which are basically sets of building instructions that can be chosen and sequenced when the build is invoked. If implemented properly, this would allow the code from multiple platforms to be compiled individually or all at once, it would allow tests to be run automatically (and perhaps different sets of tests depending on the user's choice), and would allow executable files to be generated at will. Ultimately, Ant would provide the ability to create a unified build suite for compilation, testing, packaging and friends, maximizing the convenience and effectiveness of the developers in the build phase.

Furthermore, the Travis CI tool will be used for automated testing with Git integration. When synced with Github, Travis CI can automatically run tests and communicate test results automatically whenever a new commit is pushed. The Travis functionality is itself governed by a build file, which will take advantage of the Ant build file that will be in the Git repository. Travis CI will be incredible for ensuring that the entire development team is always up to date and informed concerning the latest effectiveness of the code.

7 Design

This section will discuss the various design decisions made concerning the design of the build system, and will provide a high level view of the build system implementation.

7.1 Concerning the unification of the application platforms

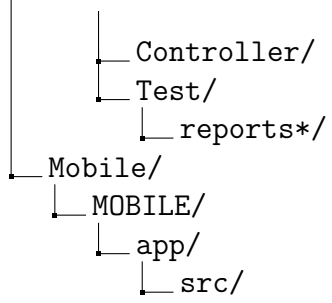
The first design decision made after deciding to use Apache Ant was to make one master build system for all platforms of the application. Although all three application platforms serve different purposes, and to the end user all platforms are literally separate, it would behoove the development team to be able to manage the build procedures for all three with the same build script. This way, should they find it appropriate, the developers may test code for more than one platform at once, for example. It is projected that this will make the build process much more efficient. Of course, the build system will be robust enough to give the user control over which platforms to target when the build is invoked. Of course, this design choice was made possible thanks to only the magic offered by Apache's Ant tool, which supports the testing protocols and compilers used for all platforms in this system.

The Build Targets section will be split up by platform and will describe the build targets for each. Note that although platform discrimination is implicit in this structure, all targets will be included in the same build file.

7.2 The calm before the storm: an overview of the intended file structure

This section will give an overview of the file structure that is inherent to the build system's design. Note that the directories and files suffixed with an asterisk denote directories and files that are generated by the build system.

```
APP/
├── build*/
│   ├── jar*/
│   │   └── <Executable Desktop JAR archive>*
│   ├── apk*/
│   │   └── <Android APK>*
│   ├── web*/
│   │   └── <Web Application zip archive>*
│   ├── desktop-classes*/
│   │   └── <Desktop Class Files>*
│   └── mobile-classes*/
│       └── <Mobile Class Files>*
├── Desktop/
│   ├── TAMAS/
│   │   ├── src/
│   │   └── test/
│   │       └── reports*/
└── Web/
    └── ca.mcgill.ecse321.group10.TAMAS/
        ├── index.php
        ├── Model/
        └── View/
```



7.3 Build targets by platform

The following section will provide a catalog of build targets structured by the target platform. Feel free to refer to directory map above when reading the paths mentioned below.

Desktop

The build targets intended for the desktop system will be enumerated below.

1. `init-desktop`
   Creates the requisite directories that will be needed for the remainder of the Desktop build targets

2. `build-desktop`
   Sources the source files from the `APP/Desktop/TAMAS/src` directory and compiles them. Puts the compiled class files in `APP/build/desktop-classes`.
   Depends on `init-desktop`.

3. `test-desktop`
   Runs all JUnit tests from the `APP/Desktop/TAMAS/test` directory and generates test reports to go to `APP/Desktop/TAMAS/test/reports`.
   Depends on `init-desktop`.

4. `export-desktop`
   Makes executable JAR archive with the class files in `APP/build/desktop-classes` directory, and places JAR in `APP/build/jar`.
   Depends on `build-desktop`.

5. `clean-desktop`
   Recursively removes all directories and files suffixed with an asterisk under the `APP/DESKTOP` directory and `build/desktop-classes` directory.

6. `run-desktop`
   Launches the desktop application, executing the executable JAR file in `APP/build/jar`.
   Depends on `export-desktop`.

Web

The following section will outline the build targets for the web application. Note that as PHP is an *interpreted* language, the code is not compiled, thus no compilation steps are included.

1. `init-web`
   Creates the `APP/build/web` and `APP/Web/ca.mcgill.ecse321.grouip10.TAMAS/Test/reports` directories.

2. `test-web`
   Runs the PHPUnit tests found in the `APP/Web/ca.mcgill.ecse321.grouip10.TAMAS/Test` directory, sends test reports to the `APP/Web/ca.mcgill.ecse321.grouip10.TAMAS/Test/reports` directory.
   Depends on `init-web`.

3. `export-web`
   Makes a zip archive of the PHP files and stores it in `APP/build/web`.

4. `run-web`
   This target will be included if it is found to be convenient. It will call a script that starts the XAMPP server and opens the `index.php` file in a browser.

Mobile

Finally, the build system design targets related to the mobile platform will be shown. Note that some of these targets depend on Desktop targets: this is because the Mobile application is heavily dependent on the logic and model of the Desktop application.

1. `init-mobile`
   Creates the directories requisite for the remainder of the mobile build targets.

2. `build-mobile`
   Compiles the source code from the `APP/Mobile/MOBILE/app/src` directory, places compiled class files in `APP/build/mobile-classes`.
   Depends on `init-mobile,export-desktop`.

3. `export-mobile`
   Builds the APK file and places it in `APP/build/apk`.
   Depends on `build-mobile`.

4. `clean-mobile`
   Removes all created directories and files (directories and files suffixed by an asterisk in the directory map) from within the `APP/mobile-classes` directory.

5. `install-mobile`
   Calls a script to install the apk file onto the connected mobile device, or to run the apk on a virtual device if no physical device is connected.
   Depends on `export-mobile`.

Convenience Targets

The following targets were designed merely for the convenience of the developers.

1. `clean-all`
   Calls `clean-desktop,clean-mobile`.

2. `build-all`
   Calls `build-desktop,build-mobile`.

3. `export-all`
   Calls `export-desktop, export-web, export-mobile`.

4. `pandemonium`
   Calls `clean-all, run-desktop, run-web, install-mobile`.

8 Rationale

Part III

Deployment

9 Description

Software deployment concerns all the tasks that cover the process of making a system available for public use. This phase will be be activated once the integrated code has tested and approved by the preceding phase. At this moment the code is ready to be released to the public, In the case of the Desktop application through a web market or physical mean,for the Mobile application through the Android marketplace or an APK sharing forum and for the web application through a a production server, such as a hosting web server to ensure its availability on the web. In this section we will go through the tools, design choices and specificities of each platform's deployment phase.

10 Tools

Within the paradigm of automatizing and avoiding manual repetitive tasks that is applied throughout the release pipeline, we look now at tools which can help with the process of deployment. These tools must be able to transfer the binaries, libraries and all other required file for the proper function of the product to each platform in a seamless, error-less automated way. All while monitoring the process, gathering metrics which can help troubleshoot any errors or improve future deployments.

The project will use Travis CI as the deployment tool for all platforms. The project's hosting provider shall be GitHub Releases. Travis CI can automatically upload assets from the Travis build directory to the team's git tags on the team's GitHub repository.
The main repository will contain a .travis.yml build file which will be used by Travis CI in order to build, automatically test and deploy the software system. The configuration for GitHub releases will be directly written in the .travis.yml file:

```
deploy:
provider: releases
api_key: "GITHUB OAUTH TOKEN"
file: "FILE TO UPLOAD"
skip_cleanup: true
on:
tags: true
```

It is highly recommended to use "travis setup releases" which will automatically create a GitHub oauth token with the correct scopes and encrypts it for deployment.

11 Design

The Deployment phase can be split into a set of well delimited activities which cover the entirety of the process. In this section we will describe them with relation to each available platform of the product.

- **Release:** Deploying GitHub Releases works only for tags, not for branches; hence the team will tag the current version of the software system if it is meant for deployment. The team will name tags that fit with semantic versioning: MAJOR.MINOR.PATCH. Every release will have a title and description. If the release is unstable, the team shall select GitHub's

"This is a pre-release" option to notify users that it's not ready for production.

Once the approved code is received from the Build Phase, the global repository containing the three platform specific applications will be tagged with an incremental numerical ID which fits with semantic versioning; this helps mark the end of the addition of a concrete sequence of fixes and upgrades to the existent applications.

The sub-repositories containing only each target platform application will then be delegated to the specific tool (addressed in the **Tools** section) for it to be deployed to the client. We will discuss this process for each platform:

- **Desktop:** The desktop application and its repository shall be released on GitHub Releases as an Open Source Software project; allowing users to fork the repository and make their own changes. The desktop program shall be released as a .jar file with the repository; so that users may access and open the program directly through their explorer. The release process will release a certain branch that contains the project the team wishes to release.
- **Mobile:** The mobile application and its repository shall be released on GitHub Releases as an Open Source Software project. The mobile application shall be packaged as an .apk (Android Application Package) file. Users will be able to fork the repository and make their own changes.
- **Web:** Because of the small size of our product, all the required libraries and executable can be found under the sub-repository containing the web application. It suffices to transfer this folder to the web server's repository. It can be achieved through the web deployment tools explained in the previous section. The repository shall also be released on GitHub Releases as an OSS project and may be forked by other users.

Any and all applications may be downloaded and packaged in a zip folder from the GitHub repository.

- **Install/Activate:** This activity is related only to **Desktop** and **Mobile** targets, as the **Web** application is accessed through the web and doesn't host any permanent or temporary files client-side.

  - **Desktop:** The `.jar` file can be placed anywhere the client wants. To avoid the need to handle OS specific file system architectures and scripting languages, once the executable is downloaded by the client the .jar will reside in the default download folder. From this point, it suffices for the client to run the executable through the method of his choice (Command Line or GUI) to access the application.
  - **Mobile:** Once the client downloads the APK the Android application handler will automatically decompress it and install it. The client requires no further action in order to access the now installed application.

- **Uninstall/Deactivate:** In the given case that one of our platforms stops being supported by the development team, a deactivation process starts. This process involves notifying the client of the end of support followed by the deactivation/uninstall of the application module.

  - **Desktop:** Uninstalling the desktop application involves deleting the .jar file of the main program and any output data that is stored in xml. If the user has cloned the repository on a local machine, then a full deactivation would require the deletion of the local repository and the user's forked repository on GitHub.
  - **Mobile:** Uninstalling the mobile application invloves uninstalling the application from the mobile phone and deletion of the .apk file by the use of a file manager for Android OS. If the user has cloned the repository on a local machine, then a full deactivation would require the deletion of the local repository and the user's forked repository on GitHub.
  - **Web:** The simplest target to deactivate, we only require to stop the web server daemon or the physical server itself.

- **Update:** Following the same logic of a full release, a subtag (i.e. x.1) would be given to the global repository containing the updates applications.

  - **Desktop:** The old `.jar` must be overwritten with the new compiled executable. If the user desires to keep an older version of the application, he simply needs to store the file under different name. Multiple versions can work concurrently.
  - **Mobile:** The android APK handler will receive the new APK and release a notification to all users who have the older version installed.
  - **Web:** Through a blue-green deployment procedure, one of the servers web daemon will he halted, the necessary configuration or script files updated and the service restarted. Once the application is up online again without problem the green server will be taken down to pass through the same procedure. Hence no downtime will be perceivable client-side

- **Version Tracking:** ersions of the release will be tracked by tags on GitHub. Since GitHub Releases requires the release to be tagged, each tag will contain all Patches, Minor, and/or Major updates specific to that software version. Each version and its changes can be viewed within the GitHub repository. Every release will be accompanied by release notes that specify the rationale and description of the release version. Along with the release notes are the commit number identifier, the time since the update has been released, the software version number, and download options (zip, and tar). In order to provide a broader coverage of our application for the customers, an archive website will be kept from where all past versions of the application can be stored and retrieved from. This log of past versions will be linked to the GitHub repository containing the open-source code of the application. In addition to this link, GitHub provides the functionality of publishing a compressed archive version of the current code. This useful feauture will be used to maintain an accessible, visible release of the latest and past versions of the application in all its target states.

- **Monitoring:** Based on the aforementioned monitoring tools, we would keep an extensive log of the current amount of installed applications on all platforms, including their up-time, their error-logs and their amount of transactions. This would help identify and prioritize the requirements for the next iteration of development. In this section of the deployment phase we enter a more continuous analysis of the production application's state. Log analysis tools such as loggly, Splunk, logstash among others would help retrieve useful, tangible metrics on the deployed applications.

12 Rationale

Why all this process: As with any phase of the Release pipeline, the design choices for the Deployment are based on the goal of achieving an automated release where the versioning, archiving and delivery of the application produces minimal errors, if not none.

Desktop Deployment: LESKO

Mobile Deployment: LESKO

Web Deployment: Choosing approaches such as the blue-green technique for the web application can be backed by security reasons:

- Disaster Recovery in case a major bug went undetected through the Integration and Build Phases.
- Reducing, or even nullifying the downtime of a web service to the clients
- Being able to mantain multiple versions of an application on separate web-servers to ensure a broader support of client-side configurations

Monitoring: The main difference with the other two phases comes from the support tasks related with the monitoring of the delivered applications. It is a continuous process that's doesn't stop since the first public release of the application is made. This part of the Deployment is intrinsic to discover faults or new requirements that appears with the continuous use of the application. As such, the usage analysis tools for big log files comes as a natural automatisation step. The metrics gathered by these applications, as well as by the Android marketplace and GitHub itself ensure a better understanding of the usage, improvements and fall-pits of the product as sa whole.