Release Builder: Automating Code Build and Deployment for Large Software Projects

Author: Lemin Li
MSc Financial Computing
Supervisor: Philip Bullett (Markit)
Chris Clack (UCL)

Submission date: 15th September 2010

This report is submitted as part requirement for the MSc degree in Computer Science at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

The Evaluated Bonds team at Markit Group Ltd develops many large software projects, the release process is currently very manual which leads to problems of human errors and low productivity. The aim of Release Builder is to better manage releases and reduce human errors by automating the code build and deployment process. Release Builder comprises of a scraper, a tagger, a builder, a deployer and a web interface that displays the progress of the release for users. Each of these sub-components specifically encapsulates an element of the release process.

In this paper we will address the challenge of designing each sub-component and how they are aligned together in a web interface. We will also look at the implementation of all the functionalities we designed. To illustrate the usage of Release Builder, screenshots of performing a release for the Evaluated Bonds project is also presented.

Acknowledgement

This project is completed with the great help of many people and I would like to give my most sincere thanks to all of them

I would like to thank my project supervisor Philip Bullett and my team members Chris Beach and Dmitri Grabov in Markit for their great help. Their guidance, support and knowledge have great influence to me and made this project an enjoyable experience. I would also like to thank Chris Clack and Robert J. Momber for their patience and their valuable advices.

I would also take this chance to thank my parents for their long lasting support and my friends for their encouragement.

Contents

Chapter 1 Introduction

This section gives an overview of the release process and how it relates to the project. The aims and objectives of the project will be described in detail and a brief overview of the structure of the report will be given.

1.1 Setting the scene

Release management is the management of the release cycle within a software project which is the distribution of software code, documentation and support materials for clients or colleagues at work to use. The role of the release manager includes managing what went into the release and where the release went, why it went there and how to deal with it when bugs are reported.

Every successful software product has multiple releases, some for internal QA[1] purposes, some are PROD[2] release for client usage. The large amount of releases for a single software product makes it necessary to make release procedure standardized and foolproof.

Building and delivering software is the main activity in a release cycle. It means the compilation or aggregation of sources into a usable utility or application. With software projects becoming larger and more complex, the demand for well-managed Build and Release functions is greater.

However, the release process for the Evaluated Bonds team at Markit Group Ltd is very manual and prone to many of problems. Firstly, human errors are common and may result in a build that is not functioning correctly, termed as bad build. Secondly, it is time consuming and involves redundant tasks.

To solve these problems and to manage the release better, we came up with the idea of a release build automation tool. The aim of which is to automate code build and deploy to standardize the release process.

1.2 Aims and objectives

The objectives of this project can be summarized as follows:

1. Manage the software release process better by streamlining the build and deployment components.
2. Minimize human error in the release process, reduce risks of bad builds thus improving product quality.
3. Save time and reduce redundant tasks for release managers.

1.3 Report structure

Chapter 2 has two sections. The first section gives a detailed description of current release process in Markit and the business requirements. The second section gives an introduction of technical background which includes the source control concepts, the implementation of subversion and the GWT framework which will be applied in development cycle.

[1] QA: Quality Assurance

[2] PROD: Production

Chapter 3 will discuss the design aspects and processes of the project and how sub-projects are linked to each other and integrated into one. It will also give some details of the interesting parts in the design.

Chapter 4 explains how the project is implemented as well as the difficulties we encounter in the implementation process and how we solved them.

Chapter 5 describes two types of tests we carried out. One is Junit tests for each sub-project, another is the whole release tests for the whole project.

Chapter 6 will present the results and outputs of the project. Screenshots of the web interface will be included also.

Chapter 7 will conclude the report by evaluating on the project and giving a detailed extensions for future development of this project.

Chapter 2 Background

This chapter gives business and technical background information about the project. In the first section, we will discuss the current software release process in EVB team and analyze the existing problems. The second section will give a brief introduction of software we used in development, source control concept and the GWT framework.

2.1 Business Background

2.1.1 Current release process

The release procedure in Evaluated Bonds team is as follows:

- Release start
  Bonds' developer will send email to announce the start of the release.
- Database scripts
  DBA[3] to run pre-packaged Database Definition Language(DDL) scripts.
  DBA to send email announcing successful completion of scripts.
- Code build
  Developers build code
- Linux installation
  Linux SA[4] to stop all applications
  Linux SA to install RPMs packages
- Web server box installation
  Linux SA to install RPM files on web server
- Windows Installation
  Windows SA[5] to install RPM files on windows
- Execute database scripts
  DBA to apply post-build DDLs[6]
- Code release testing
  Evaluated Bonds tester to confirm the release has completed as expected.

2.1.2 Manual release problem analysis

- Human error
  In the building and releasing process, human error is not an acceptable risk especially for a large software project. The EVB team develops many interoperable applications and the applications have a large amount of sub-component projects. During previous release, we encountered a lot of problems by manually building and deploying, among which the most serious one is bad build caused by human error. Release managers could forget the sequence and build packages in the wrong order or they could forget to update some component's version number. In both cases, they have to do the release all over again. Since there are so many components and release managers do repetitive work for each component, human errors are quite common. As the process is algorithmic, it could be automated.

[3] DBA: Database Administrator
[4] Linux SA: Linux System Administrator
[5] Windows SA: Windows System Administrator
[6] DDL: Data Definition Language

- Time consuming
  During code build process, release managers have to type command line by line for each component and wait until it is done. Besides, a whole release process takes long time itself which adds to the overall release time. If we could automate the process, then release managers could compress more activity into available time thus increasing their productivity.

- Low productivity
  The repetitive and redundant work as well as the human errors lead to low productivity which greatly affect the business. Automation tool could reuse modules within and across projects including tagging, building, deployment which will lead to higher efficiency.

2.2 Technical Background

2.2.1 Software used in Development

- Eclipse

Eclipse is a multi-language software development environment comprising an IDE[7] and an extensible plug-in system.

- Subversion

Subversion is free and open-source version control system. Subversion manages files and directories and the changes made to them. This allows users to track the history of how data changed and recover older versions of data.

- Confluence

Confluence is a web-based corporate wiki developed by Atlassian. Wiki refers to a website that allows users to create and edit internal linked web pages via a web browser using a simplified markup language. Confluence is written in Java and mainly used in corporate environment for knowledge management and documents.

- Jira

Jira is an issue tracking tool developed by Atlassian. It is widely used in bug tracking, issue tracking and project management. Jira is written in Java and it integrates with source control tool such as Subversion. A Jira ticket means an issue and it is raised when the issue needs to be solved.

- Junit

Junit is a unit testing framework to write repeatable tests for the Java programming language.

- UML

UML[8] is a standardized modelling language in software engineering. It is used to specify, visualized, modify, construct and document the artifacts of an object-oriented software system under development. An artifact is one of many kinds of tangible secondary product produced during the development cycle of software, for example, requirements, use cases and class diagrams. It helps describing functions, architecture and design of a software.

2.2.2 Source control

[7] IDE: Integrated Development Environment

[8] UML: Unified Modelling Language

Source control, is the management of changes to documents, programs, and other information stored as computer files. It is most commonly used in software development where a team of people may change the same files. Changes are usually identified by a number or letter code, termed the “revision number”, “revision level”. Each revision is associated with a timestamp and the person making the change. Revision can be compared, restored and with some types of files, merged.

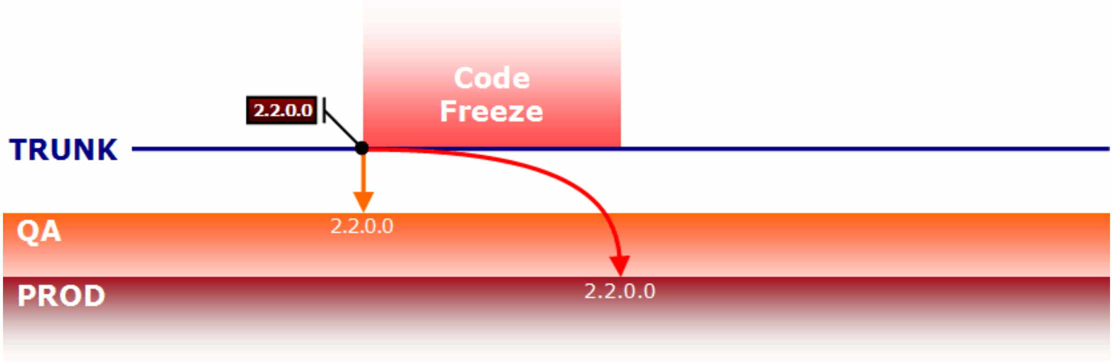
Now, we will explain some of the main vocabulary in source control:

- Trunk: The main line of development
- Branch: A set of files under source control may be branched or forked at a point in time so that, from that time forward, two copies of those files may be develop independently of each other
- Tag: refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a meaningful name or revision number. Usually, when a tag is made, it is fixed.
- Repository: The repository is where files’ current and historical data are stored, often on a server.
- Checkout: A check-out creates a local working copy from the repository. A user may specify a specific revision or obtain the latest.
- Commit: A commit occurs when writing or merging a copy of the changes made to the working copy into the repository.
- Merge: A merge or integration is an operation in which two sets of changes are applied to a file or set of files. Some sample scenarios are as follows:

Source control is an important concept to our project for two reasons: First, due to the fact that we break the projects into sub-components and allow different developers to co-work on the different sub projects at the same time, source control is extremely important to track each other’s edits, correct mistakes. Further, source control is the central concept used in the release cycle. We will explain it in detail using following examples and graphs:

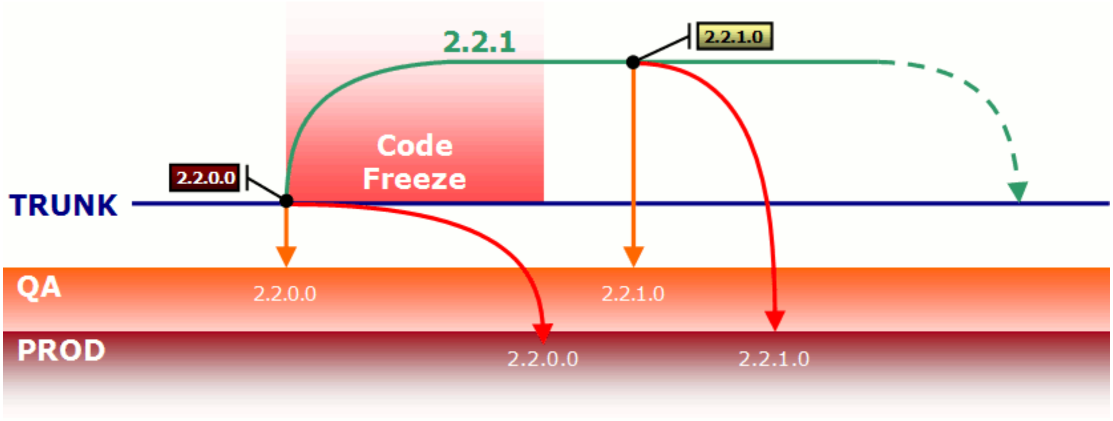
Case 1: Normal release cycle

- A tag (e.g. “2.2.0.0”) should be created when releasing to QA
- When the QA release is signed off, the same tag should be released to production
- If changes are required after the initial QA release, perform the changes on a branch (e.g. “2.2.0”) and create a second tag (e.g. “2.2.0.1”)



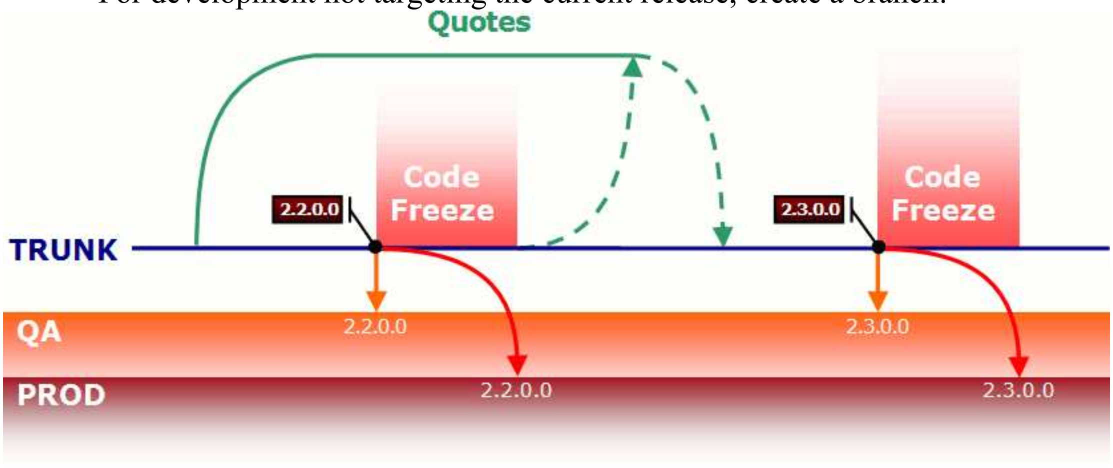
Case 2: Hot fix

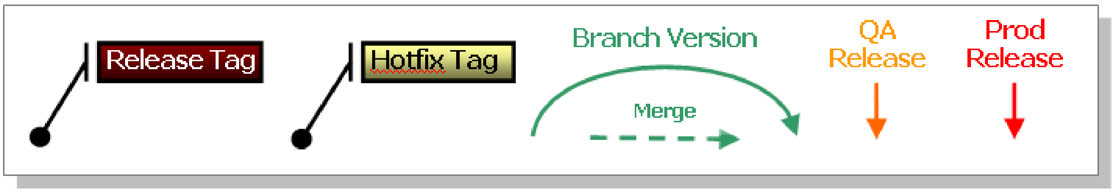
- Take a branch (e.g. “2.2.1”) from the released tag (e.g. “2.2.0.0”) and create a new tag (e.g. “2.2.1.0”) to deploy



Case 3: Long term development

- For development not targeting the current release, create a branch:





2.2.3 GWT framework

The Google Web Toolkit (GWT), is a development toolkit for building and optimizing complex browser-based applications. Its goal is to enable developers to efficiently develop high-performance web applications by creating and maintaining complex JavaScript front-end applications in Java.

The history of GWT could be dated back to May 16, 2006, GWT versions 1.0 RC 1(build 1.0.20) was released and Google announced GWT at the JavaOne conference. Since then, it has released 13 versions until now, the latest version is GWT 2.0.4 which was released June 30, 2010.

With GWT, developers can use the Java programming language to develop and debug AJAX applications and the GWT cross-compiler will translate the Java code to standalone JavaScript which can automatically run on major web browsers. Here, we listed some of the main features of GWT:

- Great development tools via Eclipse: refactoring, navigation
- IDE support: Eclipse plug-in helps with various coding tasks and errors
- Communication with servers using simple RPC mechanism
- Dynamic reusable UI components
- Junit test integration.

The first two features indicates that we can use Java, a good object-oriented language and Eclipse to develop, maintain and test codes. Eclipse, a software development environment comprising an integrated development environment and an extensible plug-in system will helps developers to write any desired extension to the environment. Java support is provided in the Eclipse SDK, with Subversion support provided by third-party plug-ins. Further more, the Eclipse SDK includes the Eclipse Java Development Tools offering an IDE with a built-in incremental Java compiler and a full model of the java source files which allows for advanced refactoring and code analysis.

Remote procedure call (RPC) is the mechanism for interacting between the client and server side of an application across a network. Within the client code, an automatically-generated proxy class is generated to make calls to the service and the serialized Java objects which are arguments in the method will be passed back and forth between client and server. There is one important fact to be noticed or RPC, that is all RPC is asynchronous. Developers have no option to implement a synchronous call to the server. For this reason, we have to create another client interface, an asynchronous one based on the original service interface. The asynchronous method calls requires the caller to pass in a callback object that can be notified when an asynchronous call completes. The server-side code that gets invoked from the client is often referred to as a service, so the remote procedure call is called invoking a service.

RPC makes it easy for the client and server components of web application to exchange Java objects over HTTP. This mechanism is able to pass and move all of the user interface logic to the client that leads to greatly improved performance, reduced web server load.

In terms of reusable UI components feature, programmers can use pre-designed classes to implement time-consuming dynamic behaviours. They can also create reusable Widget through the synthesis of other Widget, and display them easily on panel.

To ensures the quality of the web application over its lifecycle, GWT integrated Junit test. Programmers can run unit test in debugger and browser and even test asynchronous RPC.

Google Web Toolkit framework is considered to be the best to use for the web interface of our project. First of all, GWT allows us to develop using Eclipse to build the web interface. Second, all the network operations in GWT are asynchronous or non-blocking. In our web interface, we expected the progress of each component is displayed one by one in sequence instead of being showed at once when all components complete. This makes the application seem more dynamic to the users. Moreover, with the early identification of each component’s progress, the next process could start as soon as one component finishes the previous one. In this way, the asynchronous property of GWT applications perfectly matches our requirements.

Chapter 3 Design

We begin in this chapter by gathering requirements of the design. We then move on to section 3.2 where we discuss the purpose and design of standardized API. The design of sub-projects like tagging, building, deployment will be described briefly in section 3.3 to 3.6 respectively. We will finish the chapter with the design of the web interface which integrates all the sub-projects in one interface and allows users to interact with it.

3.1 Requirements and Specification

As I described in chapter 2, the tagging, building, deployment processes for different components are manually completed by release manager. Many problems occur during each release, such as human error, bad builds led by that and waste of time, so the main requirement is to reduce human errors automating the release process.

The whole release consists of three elements. First is to scrape components' information from an internal wiki page of EVB team; second is to tag components; third is to build RPM files and deploy them to destination hosts, therefore we decided to break the project into sub-projects of scraper, tagger, builder, deployer.

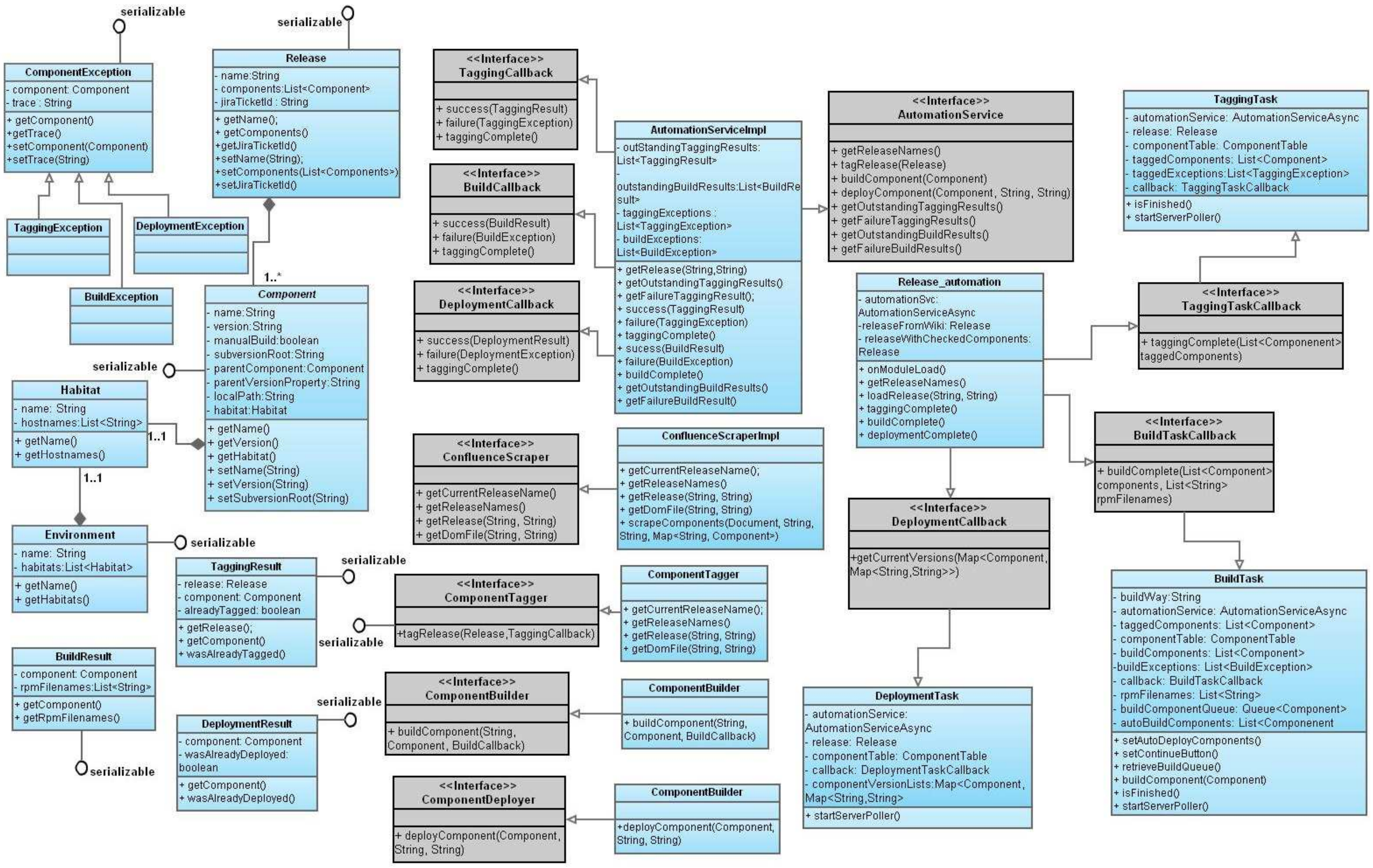
The process of scraping, tagging, building and deploying should be visualized in a web interface which will be able to perform all the functionalities of the sub-projects in sequence, as a result we decided to have a web interface for the whole release process allowing users to interact with it. We will also include standardized API of the objects classes, calling conventions and specification of methods to communicate between different sub-components.

The requirements of different sub-project are listed as below:

- For scraper, it will be able to extract all the components information including component name, version number corresponding to release and deployment hosts from Confluence page, then create release manifest XML to be used in further steps in case that confluence page goes down.
- The next step is tagging, the tagger will be able to check out the appropriate code from trunk or branch, update version numbers in pom.xml files as required and then check the projects into SVN using the appropriate tag.
- After that, the builder and deployer should be able to build RPMs and deploy RPMs to the destination hosts using SCP and use SSH to connect to the destination hosts in which process it will stop services, deploy the RPMs, start services again and check RPMs into Subversion.
- All the processes will be reflected in a web interface allowing user to choose release version and deployment environment and then the interface will be also to perform all the functionalities of scraper, tagger, builder and deployer in order and at the meantime display the progress of each component during every process.

According to the requirements, we first modelled what the interacting elements of a system will be, with each object representing an entity of importance in the system using Unified Modelling Language(UML). The class diagram is shown below and we will justify the design by explaining each class in detail in the following sections.

Class Diagram


serializable
ComponentException
- component: Component
- trace : String
+ getComponent()
+getTrace()
+setComponent(Component)
+setTrace(String)
TaggingException
DeploymentException
BuildException
serializable
Release
- name:String
- components:List<Component>
- jiraTicketId : String
+ getName();
+ getComponents()
+getJiraTicketId()
+setName(String);
+setComponents(List<Components>)
+setJiraTicketId()
1..*
Component
- name:String
- version:String
- manualBuild:boolean
- subversionRoot:String
- parentComponent:Component
- parentVersionProperty:String
- localPath:String
- habitat:Habitat
+ getName()
+ getVersion()
+ getHabitat()
+ setName(String)
+ setVersion(String)
+ setSubversionRoot(String)
serializable
Habitat
- name: String
- hostnames:List<String>
+ getName()
+ getHostnames()
1..1
1..1
Environment
- name: String
- habitats:List<Habitat>
+ getName()
+ getHabitats()
serializable
TaggingResult
- release: Release
- component: Component
- alreadyTagged: boolean
+ getRelease();
+ getComponent()
+ wasAlreadyTagged()
serializable
serializable
BuildResult
- component: Component
- rpmFilenames:List<String>
+ getComponent()
+ getRpmFilenames()
serializable
DeploymentResult
- component: Component
- wasAlreadyDeployed: boolean
+ getComponent()
+ wasAlreadyDeployed()
serializable
<<Interface>>
TaggingCallback
+ success(TaggingResult)
+ failure(TaggingException)
+ taggingComplete()
<<Interface>>
BuildCallback
+ success(BuildResult)
+ failure(BuildException)
+ taggingComplete()
<<Interface>>
DeploymentCallback
+ success(DeploymentResult)
+ failure(DeploymentException)
+ taggingComplete()
<<Interface>>
ConfluenceScraper
+ getCurrentReleaseName()
+ getReleaseNames()
+ getRelease(String, String)
+ getDomFile(String, String)
<<Interface>>
ComponentTagger
+tagRelease(Release,TaggingCallback)
<<Interface>>
ComponentBuilder
+ buildComponent(String, Component, BuildCallback)
<<Interface>>
ComponentDeployer
+ deployComponent(Component, String, String)
AutomationServiceImpl
- outStandingTaggingResults: List<TaggingResult>
- outstandingBuildResults:List<BuildResult>
- taggingExceptions : List<TaggingException>
- buildExceptions: List<BuildException>
+ getRelease(String,String)
+ getOutstandingTaggingResults()
+ getFailureTaggingResult();
+ success(TaggingResult)
+ failure(TaggingException)
+ taggingComplete()
+ sucess(BuildResult)
+ failure(BuildException)
+ buildComplete()
+ getOutstandingBuildResults()
+ getFailureBuildResult()
ConfluenceScraperImpl
+ getCurrentReleaseName();
+ getReleaseNames()
+ getRelease(String, String)
+ getDomFile(String, String)
+ scrapeComponents(Document, String, String, Map<String, Component>)
ComponentTagger
+ getCurrentReleaseName();
+ getReleaseNames()
+ getRelease(String, String)
+ getDomFile(String, String)
ComponentBuilder
+ buildComponent(String, Component, BuildCallback)
ComponentBuilder
+deployComponent(Component, String, String)
<<Interface>>
AutomationService
+ getReleaseNames()
+ tagRelease(Release)
+ buildComponent(Component)
+ deployComponent(Component, String, String)
+ getOutstandingTaggingResults()
+ getFailureTaggingResults()
+ getOutstandingBuildResults()
+ getFailureBuildResults()
Release_automation
- automationSvc: AutomationServiceAsync
-releaseFromWiki: Release
- releaseWithCheckedComponents: Release
+ onModuleLoad()
+ getReleaseNames()
+ loadRelease(String, String)
+ taggingComplete()
+ buildComplete()
+ deploymentComplete()
<<Interface>>
DeploymentCallback
+getCurrentVersions(Map<Component, Map<String,String>>)
DeploymentTask
- automationService: AutomationServiceAsync
- release: Release
- componentTable: ComponentTable
- callback: DeploymentTaskCallback
- componentVersionLists:Map<Component, Map<String,String>
+ startServerPoller()
TaggingTask
- automationService: AutomationServiceAsync
- release: Release
- componentTable: ComponentTable
- taggedComponents: List<Component>
- taggedExceptions:List<TaggingException>
- callback: TaggingTaskCallback
+ isFinished()
+ startServerPoller()
<<Interface>>
TaggingTaskCallback
+ taggingComplete(List<Componenent> taggedComponents)
<<Interface>>
BuildTaskCallback
+ buildComplete(List<Component> components, List<String> rpmFilenames)
BuildTask
- buildWay:String
- automationService: AutomationServiceAsync
- taggedComponents: List<Component>
- componentTable: ComponentTable
- buildComponents: List<Component>
-buildExceptions: List<BuildException>
- callback: BuildTaskCallback
- rpmFilenames: List<String>
- buildComponentQueue: Queue<Component>
- autoBuildComponents: List<Componenent
+ setAutoDeployComponents()
+ setContinueButton()
+ retrieveBuildQueue()
+ buildComponent(Component)
+ isFinished()
+ startServerPoller()

create and share your own diagrams at gliffy.com

3.2 Standard API

The application programming interface (API) describes a set of class definitions and functions associated with classes. As we break our project into small sub-projects, to facilitate interaction between the sub-projects, we decided to create a standardized API to specify different objects, naming conventions and methods.

In terms of objects, we have Model package to represent the object class. It includes Component, Release, Environment and Habitat class. We also have Exception package consisting of taggingException, buildingException. To define the behaviours and functions of building tools, we have the API for Scraper, Tagger, Builder and Deployer. Besides, we also include callbacks and results API such as TaggingResult, BuildResult for passing results to web user interface.

3.2.1 Model

The primary concept in the project is release. Each release object is identified by its name. It also includes a list of sub-projects known as components. Each release is associated with an assigned Jira ticket for source control purpose.

The component object in release has basic attributes of name, version and deployment habitat. The version number is uniquely linked to a release name of this component. The deployment habitat specifies the server on which the deployment will take place. However, there are two types of components, one does not need to be deployed but only act as dependencies of other components. These components only have the basic three attributes while another type of components have more attributes. Due to the fact that some components are dependent on the parent project, some more attributes must be incorporated into the components such as parent project, version property of parent project, subversion root to record the correct information to do tagging, building and deployment. Thus the component object has two constructors, one only takes arguments of the three basic attributes, one with the additional attributes including subversion root, parent component, parent version property and whether it is manually built.

Environment is another object we incorporated into model package. Environment has names of QA, DEV or PROD which indicates different release environments. It also contains a list of habitats which are identified by its name and corresponding servers' hostnames.

3.2.2 Exceptions

Tagging, building or deploying, a component is not an infallible operation. So we defined exceptions to record detailed information about exceptions thrown in this process. Since all the exceptions follow the same structure, we defined a ComponentException class to standardize the exception and all the TaggingException, BuildExcetpion and DeploymentExcetpion will extend the standard ComponentException.

The ComponentException itself extends Exception class, so it gets the error message. So the constructors in exception classes can take the argument of the error message and also the component causing the exceptions.

3.2.3 Build tools

In order to abstract the behaviours of building tools, we defined build tool API. In scraper, the main function is to get release, so we defined getRelease() method. In terms of tagger, we defined tagRelease() to tag components in a release. Builder is slightly different, since some of components should be manually built at this stage, instead of building all the components at once, we built component one by one to builder. It works the same for the deployer, once a component is built, it will be immediately passed to deployer. Subsequently, we defined the main method in builder and deployer as buildComponent(Component) and deployComponent(Component).

3.2.4 Results and callbacks

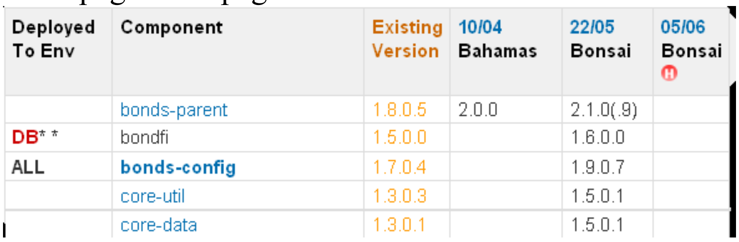
To pass results to web user interface and display the result information to users, we decided to have a result and callback API.

Callback is a reference to executable code which is passed as an argument to other code. A callback is a subtle way to pass the results and signal exceptions and errors without stopping the process. For example, in the tagging or building process, we did not want the program stop after it gets errors but just give a failure signal, to make sure the user is notified, it will register the exception's information as a buildCallback. For example, in TaggingCallback class, we used a method called Success which takes the TaggingResult as argument, Failure method which takes TaggingException as argument as well as taggingComplete to indicate the tagging process is finished.

In terms of results, each result returned in the process is the component which has been successfully built and should have the information of which release it relates to, which component it is and some additional process-specific information. For example, in Tagger, the TaggingResult constructor takes in the release, the component and whether this component is already tagged.

3.3 Confluence Scraper

The first step of automating release process is to scrape the information of all the components for the software release. For every release, the components' attributes including corresponding version number and deployment servers will be listed in team's internal wiki page. This page holds a table as below:



Deployed To Env	Component	Existing Version	10/04 Bahamas	22/05 Bonsai	05/06 Bonsai Ⓗ
	bonds-parent	1.8.0.5	2.0.0	2.1.0(.9)	
DB* *	bondfi	1.5.0.0		1.6.0.0	
ALL	**bonds-config**	1.7.0.4		1.9.0.7	
	core-util	1.3.0.3		1.5.0.1	
	core-data	1.3.0.1		1.5.0.1	

Originally, release manager had to find each component in the repository one by one by referencing the confluence page. The build automation will be able to solve this problem by extracting all the information automatically from the page.

The design for scraper is quite simple and straightforward. The tool will take a release name such as "Cuba" as an argument and identifies:

1. The relevant column on the page
2. All the components to be included in the release
3. Component versions

4. Component deployment hosts
5. Other attributes such as subversion root, parent project that will affect the process

By specifying the above requirements, we defined several classes specifying different functionalities in the confluence scraper tool. The first one is the utility tool for scraper. It includes a web page fetcher which can get the contents of web page, parse it to document object. It also consists of cell text extractor tool which can extract the text of any cell in table. Based on the first tool, we then defined release object generator which creates a release object by retrieving the correspondent data information of components using the first tool.

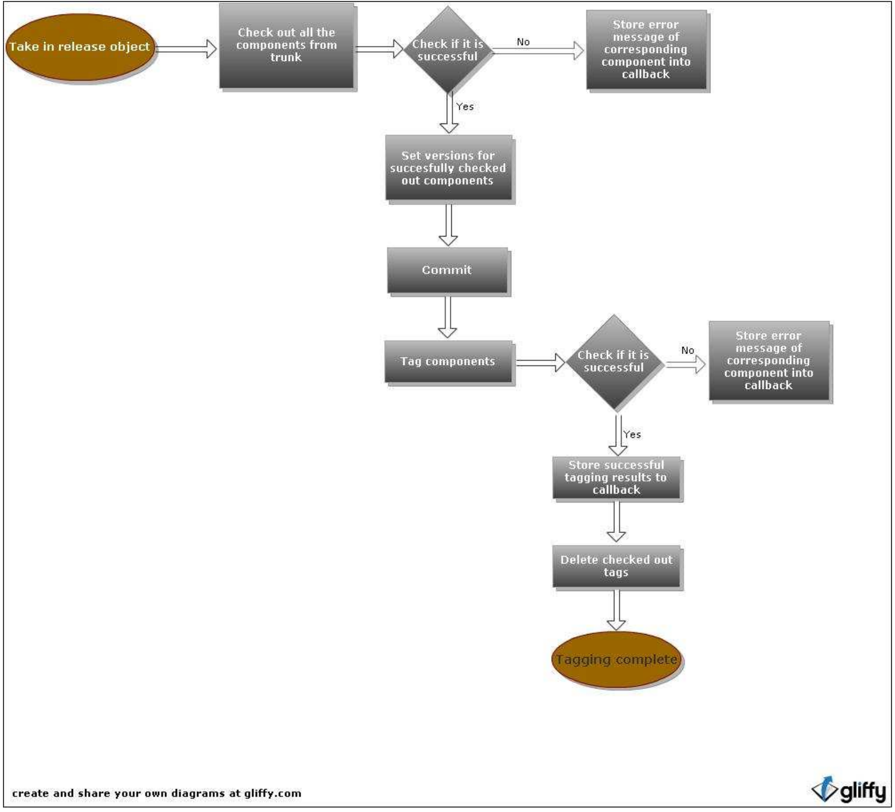
The methods in ScraperUtil are those shared by ComponentScraper and ServiceScraper. To make the structure as simple and standard as possible, we defined the basic web fetch method getDocument() and cell text extraction method getCellText as static methods in ScraperUtil class so that ComponentScraper and ServiceScraper can use it directly without generating new object.

The getDocument() method is defined to fetch the content of different web page identified by the URL string passed as an argument and parse it to document and return it. After that, the getCellText() method takes arguments of the document and table number, column number as well as row number defining which cell's text you wish to extract. The methods getRowNumber() and getColumnNumber() are defined to look up for specific row and column given a text.

3.4 Component Tagger

Tagger is the tool to check out the appropriate code from trunk or branch , update version numbers in pom.xml files as required and then check the projects into Subversion using the appropriate tag.

The work flow for tagger is showed as below:


Take in release object
Check out all the components from trunk
Check if it is successful
No
Store error message of corresponding component into callback
Yes
Set versions for succesfully checked out components
Commit
Tag components
Check if it is successful
No
Store error message of corresponding component into callback
Yes
Store successful tagging results to callback
Delete checked out tags
Tagging complete
create and share your own diagrams at gliffy.com
gliffy

First check out all component projects from trunk, then set versions for all component projects except bond-parent in the POM, commit it then. The next step is to tag. The tag/branch logic works as follows:

- For versions ending with .0 (e.g.1.2.0.0), if the 1.2.0.0 tag does not already exist
  -check out from trunk and alter POM version to 1.2.0.0
  -tag 1.2.0.0
- For versions ending in number other than 0 (e.g. 1.2.0.1) and if the tag does not already exist:
  -check for existence of branch 1.2.0 (fail if not found)
  -check out from branch and alter POM version to 1.2.0.1
  -tag 1.2.0.1

After tagging, we delete the checked out tags and the tagging is done.

3.5 Component Builder and Deployer

Two network protocols were used in component builder and deployer. Secure Copy protocol (SCP), a way of securely transferring files between local and a remote host or between two remote hosts is used in builder. Secure Shell (SSH), a network protocol that allows data to be exchanged using a secure channel between two networked devices is used to log into a remote machine and execute commands to deploy RPMs.

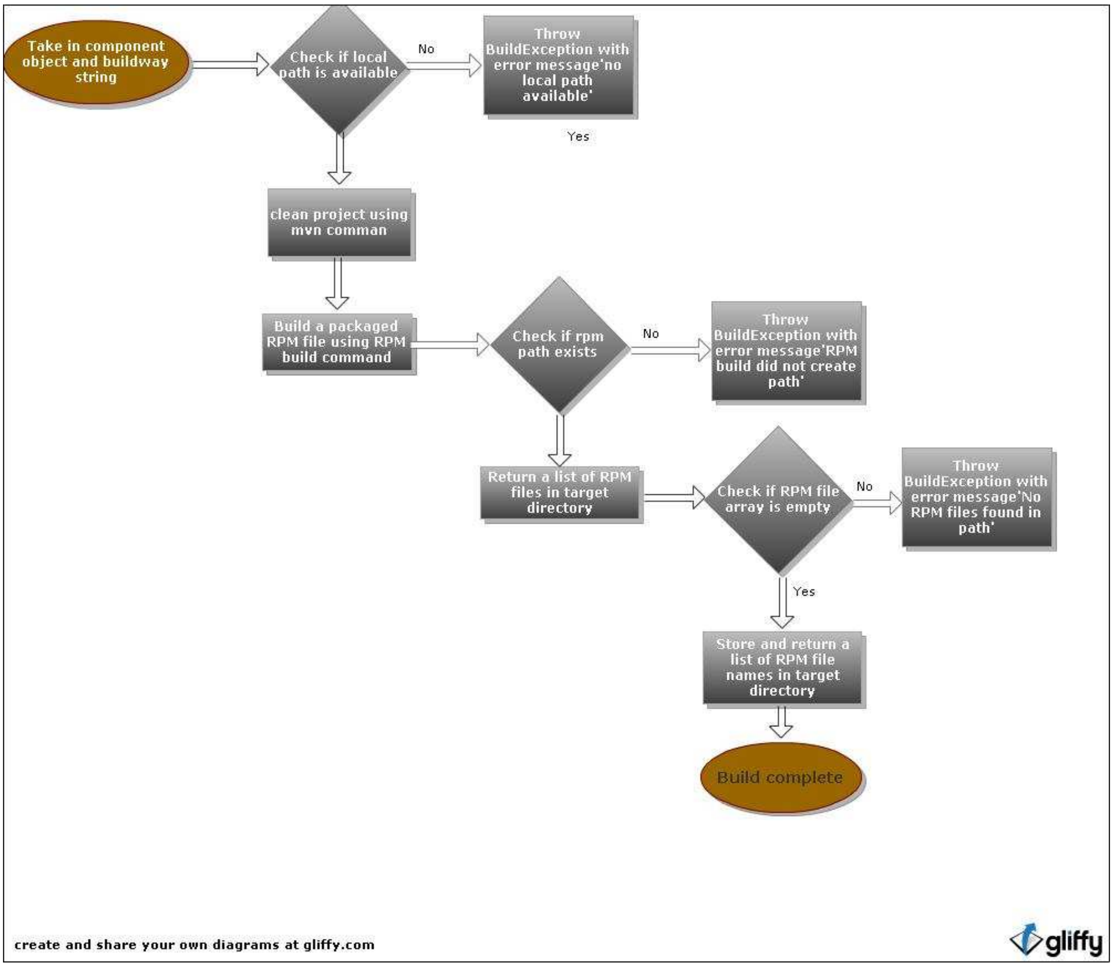
The tool builder should use the release manifest XML file generated by scraper and files checked out by tagger to:

- Build RPMs
- Deploy RPMs to the destination hosts using SCP

SCP is a way of securely transferring files between local and a remote host or between two remote hosts.

- Use SSH protocol to connect to the destination hosts:
  - Stop services
  - Deploy the RPMs
  - Start services again
  - Check RPMs into Subversion

The work flow for builder is as below:


Take in component object and buildway string
Check if local path is available
No
Throw BuildException with error message'no local path available'
Yes
clean project using mvn comman
Build a packaged RPM file using RPM build command
Check if rpm path exists
No
Throw BuildException with error message'RPM build did not create path'
Return a list of RPM files in target directory
Check if RPM file array is empty
No
Throw BuildException with error message'No RPM files found in path'
Yes
Store and return a list of RPM file names in target directory
Build complete
create and share your own diagrams at gliffy.com
gliffy

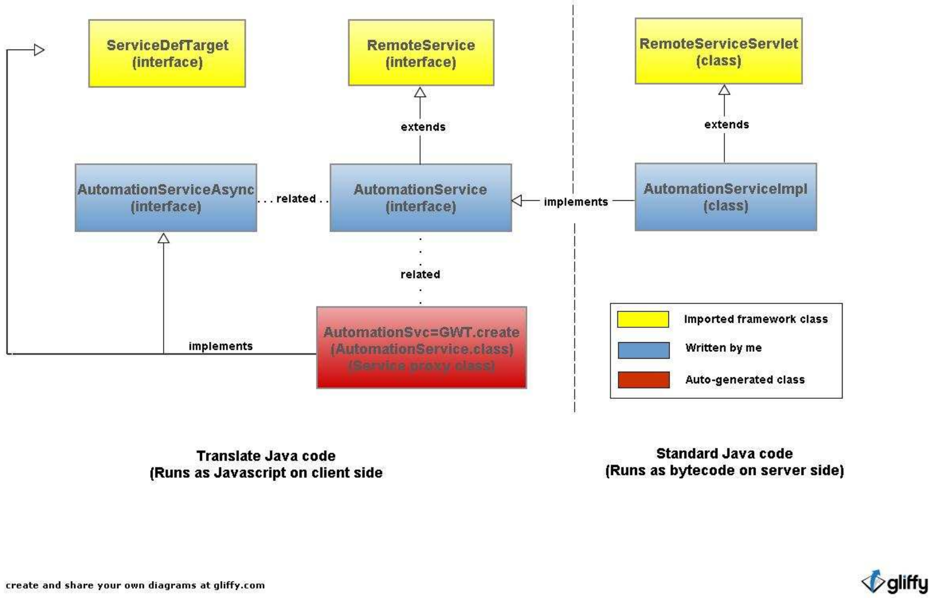
Since building and deployment are separate processes, we broke them into two sub-components builder and deployer. The key element in builder and deployer is to run command in Unix and do execute the operations.

3.6 Web Interface

All the processes we described before should be aligned together and executed in a specific order and also be displayed and interacted with users. So we need a central point to access all the processes and a user interface too. The specific core functions of this sub-component we come up with are listed as below:

- Aligning the process of tagging, building and deployment
- Displaying the progress by clearly showing error and success information for every component during each phase
- Allowing user to choose release version and deployment environment and then perform all the functionalities by calling the methods of sub-projects.
- Passing results and exceptions during each process

In order to meet the requirements, we designed this web interface leveraging Google Web Toolkit framework. According to the GWT RPC mechanism and the asynchronous feature, a diagram is showed below to illustrate how the design works.


ServiceDefTarget (interface)
RemoteService (interface)
RemoteServiceServlet (class)
extends
extends
AutomationServiceAsync (interface)
. . . related . .
AutomationService (interface)
implements
AutomationServiceImpl (class)
related
implements
AutomationSvc=GWT.create (AutomationService.class) (Service proxy class)
Imported framework class
Written by me
Auto-generated class
Translate Java code (Runs as Javascript on client side
Standard Java code (Runs as bytecode on server side)
create and share your own diagrams at gliffy.com
gliffy

- Client side: AutomationService interface

In order to define RPC interface, we defined AutomaionService that extends RemoteService interface. The reason to use this interface lies in the fact that on the client side we need to call the method to receive the response later. In the Automation Service class, we listed all the RPC methods of executing the release process like getReleaseNames(), tagRelease(Release release), buildComponent(String buildWay, Component component) as well as methods of getting results includes getOutstandingTaggingResults(), getFailureTaggingResults().

- Client side: AutomationServiceAsync interface

Before we make a remote call from the client, we created another client interface called AutomationServiceAsyn, an asynchronous interface based on the AutomationService interface. The AsyncCallback functions like an Event Listener, when the server sends the response back, we could get an event containing the success or failure data. Since an asynchronous method requires the caller to pass a callback object that can be notified when an asynchronous call completes, asynchronous methods have return types. All the communication back to the caller is via the callback object, we could see from the methods getReleaseNames(AsyncCallback<Collection<String>> callback).

- Server side: AutomationImpl

After we defined service's interface, we could design this service on the server-side. The server side class AutomationServiceImpl should extend RemoteServiceServlet for the reason that we need an HTTP path that client can send data to as well as RemoteServiceServlet decode GWT serialization and invoke the methods. The AutomationServiceImpl class also implements the AutomationService interface, so it implements all the methods in the interface such as getReleaseNames, tagRelease(Release release) and buildComponent(String buildWay, Component component).

The interesting part in our design is that AutomationImpl also implements TaggingCallback and buildCallback interface we defined in the standard API. The logic behind this is to get the results and errors from the callback of the server side, so it can pass the results and errors to client side easily. The hard part is that the results and exceptions are objects passing between servers and clients by making remote procedure calls, they must be serialized. Serialization is the process of packaging the contents of an object so that it can be passed from one application to another application or stored for later use. GWT requires all the parameters and return types to be serializable object. So we serialized the object by making ComponentException class and all the result classes implement Serializable so that all the results and exceptions can be serializable.

Once we are capable of getting the successful results or failure exception of components, we could store them in array lists and return two array lists respectively. Take Tagger for example, we defined getOutstandingBuildResults and getFailureBuildResults in RemoteServiceServlet to get lists of results and exceptions and pass back to client side.

- Client side: Release_automation

The next step is to design the client side code by creating a class called Release_automation. In this class, we aligned the tagging, building and deployment together by creating separate classes of TaggingTask and BuildingTask and calling them when needed. We then created a ComponentTable class to display the progress of each process by showing each component's status one by one in a table. Various of widgets such as ListBox, Button, TextBox are set up to help interacting with users.

- Client side: tasks

The TaggingTask and BuildTask are classes we designed to execute actual tagging and building by making RPC calls to server and getting callback of results and exceptions. Take TaggingTask as an example, it calls method tagRelease in AutomationService and gets callbacks which contains the taggingResult or taggingException from methods of getOutstandingTaggingResults() or getOutstandingTaggingException() from server side.

- Client side: component table

ComponentTable class is designed to display tables and the release progress. It extends FlexTable class thus it can use its methods directly. ComponentTable has several methods to display different tables at different stages. First, it displays a table with all components of a given release name using populateWithRelease(Release release) method. When tagging process begins, it added a column labelled "Tagged" with spinners in each component indicating all the components are waiting to be tagged using method componentStartTagged(Release release). During the tagging process, the table displays the cell each component is in with either a tick indicating the component is successfully tagged using componentHasTagged(TaggingResult taggingResult) or a red cross meaning it failed by componentTaggingFailed(TaggingException e). The building process applies the same rule.

Chapter 4 Implementation

In this chapter, we will describe the implementation of the project by discussing the problems we encountered, how we resolved them and the interesting elements in implementation. We will go through each component sub-project, and explain how we implemented the design ideas. This will start by problem analysis first, followed by describing every step involved in the implementation. In each section, we will explain some of the core codes to give more details of the implementation.

4.1 Confluence Scraper

To achieve the functions we discussed in section 3.3, we fetched HTML content of a web page first, parse it to Document and extract the required information, afterwards marshal the content to XML format.

4.1.1 Web page fetcher

To fetch the HTML content of a web page, we first created a URL object to specify the address of the web page which is the confluence page of Evaluated Bonds. For the URL object, then use openConnection method of URLConnection class to get URLConnection object. The method of openConnection is from URLConnection class which represents communications of application and URL and it can be used to read from and to write to the resource referenced by the URL. After setting general request property encoded by base 64 scheme, this URLConnection object can be used to create DataInputStream. Finally, we created BufferedReader object using InputStream object and fetched the contents line by line using readLine method of DataInputStream object.

To parse the contents of the fetched web page, we used DocumentBuilder class calling the parse method to parse the input stream. After we got the document object, we could process the contents of the doc by retrieving the information we want.

4.1.2 Release object generator

The ultimate goal of scraper is to get the release object given the argument of release name and release environment. This is also the interesting part due to the layout of the confluence page.

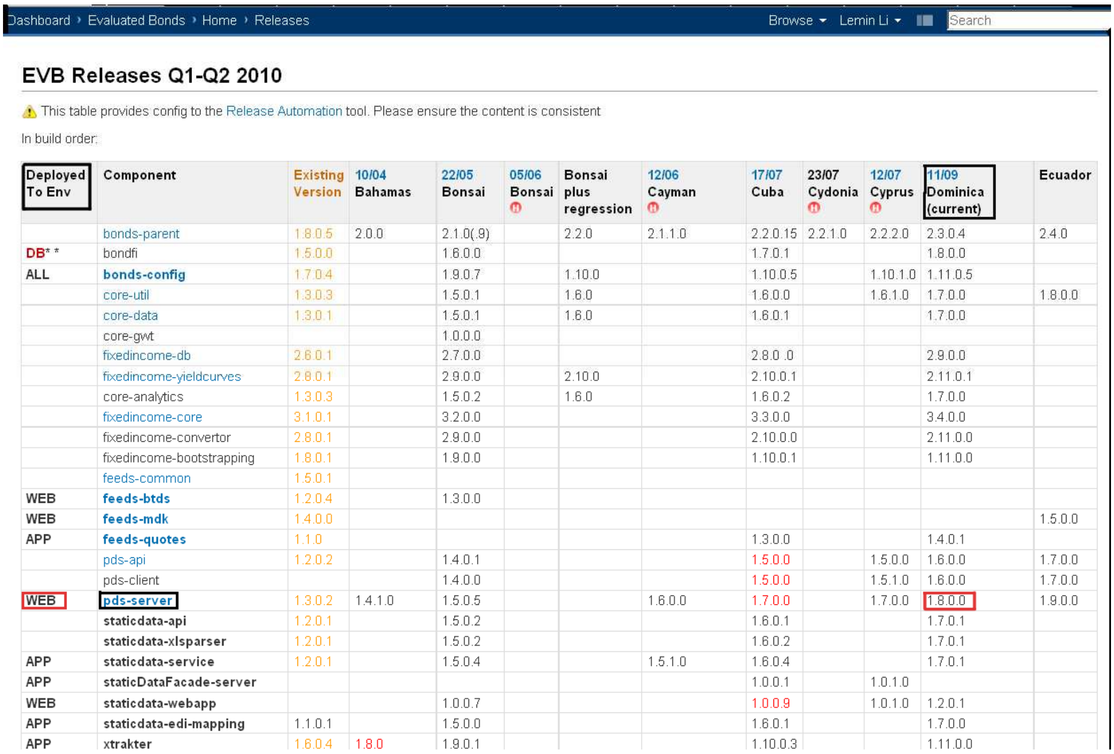
The problem to get the full information of a component is that there are two tables displayed in the confluence page that contain different information of component. One lists with all the components and their version numbers regarding to different release, one lists with the attributes of those components which are dependent on parent project. To retrieve the second table's data, we need to reference the first table.

The key is to retrieve all the information from the two tables and store them in one component. Initially, we treated the components listed in the two tables differently and stored them in two lists. However, during the first trial test, it turned out to be quite inefficient to get the complete information of components since we had to go through two methods to compare components' names and then get the information which greatly increase the time complexity of retrieval.

To solve the problem, we decided to use LinkedHashMap function to represent a hash table that associates keys and values. The reason we use LinkedHashMap is that all the components should be stored in the same order as the confluence page displays. HashMap class has no specification of orders while LinkedHashMap can.

Thus, we created a LinkedHashMap called component map in the constructor of the ConfluenceScraperImpl class. The idea is when we went through the first table, we can store all the component names and their components into the component map by using put() method. By doing this, we created a link between component names as keys and components as values. Then we could reference it for further identification and retrieval of components. After that, when we looked at the second table, we could retrieve the component using component name as key and then set the additional attributes we got from the table to it.

The table is showed below to illustrate the steps to get all the information:



EVB Releases Q1-Q2 2010

⚠ This table provides config to the Release Automation tool. Please ensure the content is consistent

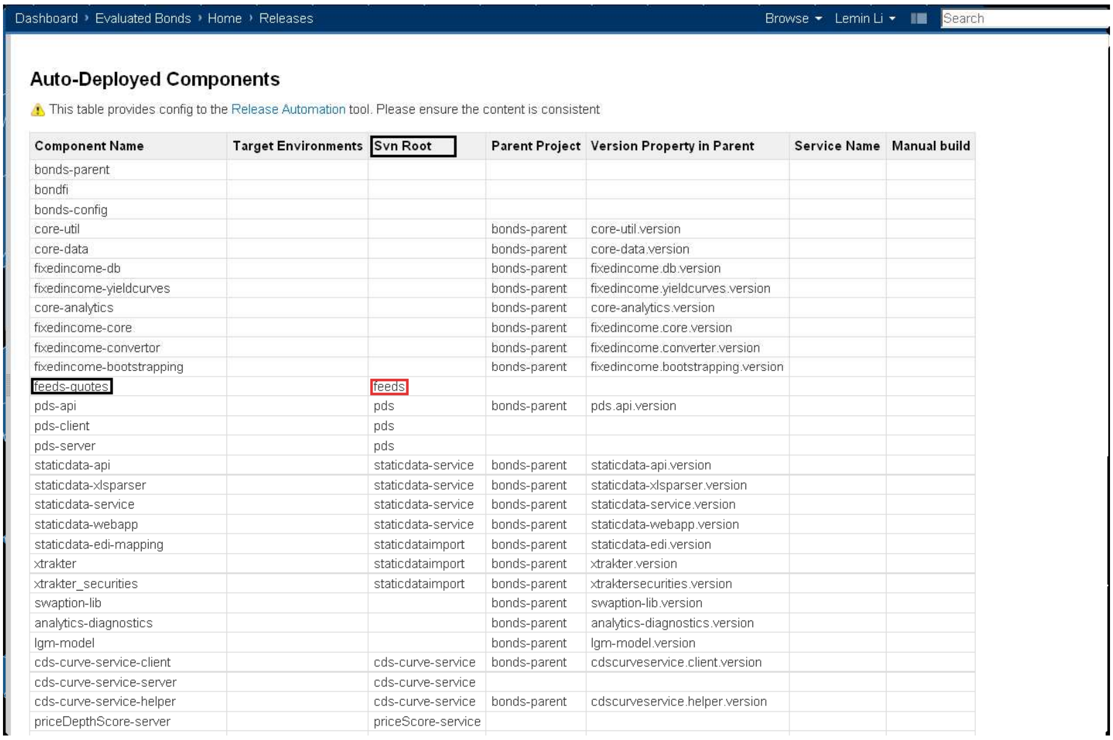
In build order:

Deployed To Env	Component	Existing Version	10/04 Bahamas	22/05 Bonsai	05/06 Bonsai	Bonsai plus regression	12/06 Cayman	17/07 Cuba	23/07 Cydonia	12/07 Cyprus	11/09 Dominica (current)	Ecuador
	bonds-parent	1.8.0.5	2.0.0	2.1.0(.9)		2.2.0	2.1.1.0	2.2.0.15	2.2.1.0	2.2.2.0	2.3.0.4	2.4.0
DB* *	bondfi	1.5.0.0		1.6.0.0				1.7.0.1			1.8.0.0	
ALL	bonds-config	1.7.0.4		1.9.0.7		1.10.0		1.10.0.5		1.10.1.0	1.11.0.5	
	core-util	1.3.0.3		1.5.0.1		1.6.0		1.6.0.0		1.6.1.0	1.7.0.0	1.8.0.0
	core-data	1.3.0.1		1.5.0.1		1.6.0		1.6.0.1			1.7.0.0	
	core-gwt			1.0.0.0								
	fixedincome-db	2.6.0.1		2.7.0.0				2.8.0 .0			2.9.0.0	
	fixedincome-yieldcurves	2.8.0.1		2.9.0.0		2.10.0		2.10.0.1			2.11.0.1	
	core-analytics	1.3.0.3		1.5.0.2		1.6.0		1.6.0.2			1.7.0.0	
	fixedincome-core	3.1.0.1		3.2.0.0				3.3.0.0			3.4.0.0	
	fixedincome-convertor	2.8.0.1		2.9.0.0				2.10.0.0			2.11.0.0	
	fixedincome-bootstrapping	1.8.0.1		1.9.0.0				1.10.0.1			1.11.0.0	
	feeds-common	1.5.0.1										
WEB	feeds-btds	1.2.0.4		1.3.0.0								
WEB	feeds-mdk	1.4.0.0										1.5.0.0
APP	feeds-quotes	1.1.0						1.3.0.0			1.4.0.1	
	pds-api	1.2.0.2		1.4.0.1				1.5.0.0		1.5.0.0	1.6.0.0	1.7.0.0
	pds-client			1.4.0.0				1.5.0.0		1.5.1.0	1.6.0.0	1.7.0.0
WEB	pds-server	1.3.0.2	1.4.1.0	1.5.0.5			1.6.0.0	1.7.0.0		1.7.0.0	1.8.0.0	1.9.0.0
	staticdata-api	1.2.0.1		1.5.0.2				1.6.0.1			1.7.0.1	
	staticdata-xlsparser	1.2.0.1		1.5.0.2				1.6.0.2			1.7.0.1	
APP	staticdata-service	1.2.0.1		1.5.0.4			1.5.1.0	1.6.0.4			1.7.0.1	
APP	staticDataFacade-server							1.0.0.1		1.0.1.0		
WEB	staticdata-webapp			1.0.0.7				1.0.0.9		1.0.1.0	1.2.0.1	
APP	staticdata-edi-mapping	1.1.0.1		1.5.0.0				1.6.0.1			1.7.0.0	
APP	xtrakter	1.6.0.4	1.8.0	1.9.0.1				1.10.0.3			1.11.0.0	

The steps to get all the information is listed as below:

- Get the column numbers of the first table of confluence page in correspondent to the release name and deployment environment passing arguments like “Dominica” or “Deployed to env” as key words.
- Go through all the components in the first table and get contents of component version and deployment environment to the corresponding column (To get the deployment environment, we scraped the server page, the working principle is the same as this one)
- Create a component one by one taking component name, version number and environment as constructor arguments
- Create a hashMap to store the components using component name as key and component as value for further retrieval of components’ information

After that, the second step should be getting the additional data of components from the fourth table which is showed as below. The principle is the same as to get information from first table.



Auto-Deployed Components

⚠ This table provides config to the Release Automation tool. Please ensure the content is consistent

Component Name	Target Environments	Svn Root	Parent Project	Version Property in Parent	Service Name	Manual build
bonds-parent						
bondfi						
bonds-config						
core-util			bonds-parent	core-util.version		
core-data			bonds-parent	core-data.version		
fixedincome-db			bonds-parent	fixedincome.db.version		
fixedincome-yieldcurves			bonds-parent	fixedincome.yieldcurves.version		
core-analytics			bonds-parent	core-analytics.version		
fixedincome-core			bonds-parent	fixedincome.core.version		
fixedincome-convertor			bonds-parent	fixedincome.converter.version		
fixedincome-bootstrapping			bonds-parent	fixedincome.bootstrapping.version		
feeds-quotes		feeds				
pds-api		pds	bonds-parent	pds.api.version		
pds-client		pds				
pds-server		pds				
staticdata-api		staticdata-service	bonds-parent	staticdata-api.version		
staticdata-xlsparser		staticdata-service	bonds-parent	staticdata-xlsparser.version		
staticdata-service		staticdata-service	bonds-parent	staticdata-service.version		
staticdata-webapp		staticdata-service	bonds-parent	staticdata-webapp.version		
staticdata-edi-mapping		staticdataimport	bonds-parent	staticdata-edi.version		
xtrakter		staticdataimport	bonds-parent	xtrakter.version		
xtrakter_securities		staticdataimport	bonds-parent	xtraktersecurities.version		
swaption-lib			bonds-parent	swaption-lib.version		
analytics-diagnostics			bonds-parent	analytics-diagnostics.version		
lgm-model			bonds-parent	lgm-model.version		
cds-curve-service-client		cds-curve-service	bonds-parent	cdscurveservice.client.version		
cds-curve-service-server		cds-curve-service				
cds-curve-service-helper		cds-curve-service	bonds-parent	cdscurveservice.helper.version		
priceDepthScore-server		priceScore-service				

To get the column numbers and to get corresponding text, we first defined a recursive method called extractTextFromChildren to extract text from children nodes of a parent node. This method allows us to get the text nodes by concatenating text nodes with the parent element and return the value of the text node by recursively appending the value to a string buffer.

Now that we got the content of any node, we can easily get the column number given a text string. The method of getColumnNumber will get the node lists of the first row of a table using getElementsByTagName(“tr”) method. Then we went through all the nodes of the node list and use extractTextFromChildren method to get content and compare the text we got and the text given to decide if the column is what we look for. If the two texts are equal, then we found the column index and return the value.

Once we are capable of getting the column number of a given text string, we can get the column text from the column number by retrieve the text content from the node correspondent to the column number. To expand this method, we also took in the arguments of row index, table index so that we could retrieve the text of any cell in any table.

4.2 Component Tagger

In implementing those, we adopted SVNKit framework to manage working copies. SVNKit is a Java toolkit for working with data versioned by Subversion version control system right within Java applications. In SVNKit architechture, all working copy functions are logically organized in different SVN client classes. There is a SVNClientManager class containing all SVN client classes which simplifies work on creating and maintaining different client classes.

As we already discussed design of tagger in section 3.2, we implemented tagger based on the work flow. The basic functionalities of tagger is outlined in pseudo code as below:

```
                             // Check out all projects
               for (Component c : components) {
                       try {
                              □pmFilen(c, false);
                       } catch (TaggingException e) {
                              callback.failure(e);
                              errorComponents.put(e.getComponent(), e);
                       }
               }
               // Set versions
               for (Component c : components) {
               if (errorComponents.containsKeyI||component.equals(bond-parent)) {
                              continue;
                   }

                       Component target = c.getParentComponent();
                        String versionXpath =
c.getParentComponent().getParentVersionProperty();
                       setVersion(target, versionXpath, c.getVersion());
               }
               // Commit
               for (Component c : components) {
                       if (errorComponents.containsKeyI) {
                              continue;
                       }
                       commit(c, jira);
               }
               // Tag
               for (Component c : components) {
                       if (errorComponents.containsKeyI) {
                              callback.failure(errorComponents.getI);
                              continue;
                       }
                       try {
                              □pmFile alreadyTagged = tag(c, jira);
                              File checkoutPath = □pmFilen(c, true);
                              if (checkoutPath == null) {
                                      throw new TaggingException(c, "No checkout path,

returned after checkout");
                              }
                              c.setLocalPath(checkoutPath.getAbsolutePath());
                              TaggingResult result = new TaggingResult(release,c,
alreadyTagged);
                              callback.success(result);
                       } catch (TaggingException e) {
                              callback.failure(e);
                       }
               }
               // Delete checked out tags
               callback.deleteCheckedOutTags();
```

In tagger, we utilized SVNClientManager class to do all the functionalities. In doing checkout, we called getUpdateClient() and then do check out by using doCheckout() of SVNUpdateClient object and then set versions. After that, we get SVNCommitClient object and commit by calling doCommit() method. After that, we could generate SVNCopyClient to do tagging by calling doCopy() function.

4.3 Component Builder and Deployer

The key element in builder and deployer is to run command in Unix environment, as long as we can connect to the command runner, we can execute all the operations. To implement it, we used Java library ProcessBuilder to run local command in builder and j2ssh framework to run remote command in deployer.

Since in deployer, we had to run remote command each time to check if rpm can be installed and then stop service, install rpm and start service again, we created a wrapper class executeRemoteCommand(hostname, username, password,command). To adapt the j2ssh framework to have a different interface. Instead of writing a long code and run it every time when we need to connect to remote command in Unix, we could just call the wrapper class and make the code simpler.

We implemented builder and deployer based on the work flow in section 3.3. Due to the large amount of code, we just outlined the basic functionalities of builder and deployer.

Builder:

```
                try {
                        if (component.getLocalPath() == null) {
                                throw new BuildException(component, “No Local path
available for component “ + component);
                        }
                        File projectRoot = new File(component.getLocalPath());
                        commandRunner.runCommand(COMMAND_MVN + buildWay, projectRoot);
                        final List<String> rpmFilenames = buildRpms(component,
projectRoot);
                        callback.success(new BuildResult(component, rpmFilenames));
                } catch (CommandException e) {
                        callback.failure(new BuildException(component, e.getMessage(),
e));
                } catch (BuildException e) {
                        callback.failure(e);
                }
```

Deployer:

```
                try {
                        File rpmFile = new File(□pmFilenames);
                        FileObject remoteRpm = RemoteFileUtil.copyFileToUrl(rpmFile,
getRemoteUrl(hostname));
                        checkRpmCanBeInstalled(remoteRpm, hostname);
                        stopService(component, hostname);
                        installRpm(remoteRpm, hostname);
                        startServive(component, hostname);
                } catch (Exception e) {
                        throw new DeploymentException(component, e.getMessage(), e);
                }
```

4.4 Web Interface

The web interface is an application for users to interact. To capture the application's behaviours requirements, we summarized use cases.

The table lists different use cases that we developed together with the alternate flows that derive from the main use cases.

Use Case ID	Use Case Name	Alternative Flow
1	DisplayCurrentRelease	
2	ChooseReleaseEnvironment	
3	ChooseReleaseName	
4	EnterJiraTicket	InvalidJiraTicket
5	ChooseBuildWay	
6	StartTasks	ManualBuild
7	Refresh	

The use case tables show the details of each use case, in particular all the different actors that interact in each case as well as the pre- and post- conditions to each case.

They also show the main flows which is essentially what happens in each case and the alternate flows that can occur during the main flow and at which step they occur.

Use Case	DisplayCurrentRelease
ID	UC1
Brief Description	The system displays the user the current release
Primary Actors	System
Secondary Actors	User
Pre conditions	The system loads the confluence page
Main flow	1. The user loads the application 2. The system displays the current release with a default release environment
Post conditions	A release is displayed
Alternative flows	None

Use Case	ChooseReleaseEnvironment
ID	UC2
Brief Description	The system displays the user the current release
Primary Actors	User
Secondary Actors	System
Pre conditions	1. A release name is chosen 2. A release environment menu is displayed
Main flow	1. The user chooses a release environment from the release environment menu 2. The system displays correspondent release with the chosen environment
Post conditions	A release is displayed
Alternative flows	None

Use Case	ChooseReleaseName
ID	UC3
Brief Description	The user chooses release name from the release name menu
Primary Actors	User
Secondary Actors	System
Pre conditions	1. A release environment is chosen 2. A release name menu is displayed
Main flow	1. The user chooses a release name from the release name menu 2. The system displays correspondent release with the chosen name
Post conditions	A new release is displayed
Alternative flows	None

Use Case	EnterJiraTicket
ID	UC4
Brief Description	The user enters Jira ticket in the text area
Primary Actors	User
Secondary Actors	System
Pre conditions	A text area for user to enter Jira ticket is displayed
Main flow	1. The user enters a Jira ticket in the text area 2. The system displays users radio buttons of build way
Post conditions	A release is displayed
Alternative flows	None

Use Case	InvalidJiraTicket
ID	UC4.1
Brief Description	The system informs the user that the Jira ticket entered is not valid
Primary Actors	System
Secondary Actors	User

Pre conditions	The system has identified the Jira ticket is not valid
Main flow	1. The alternative flow starts after step 1 of the main flow 2. The system displays an error message that the Jira ticket is invalid 3. The system returns to step 1 of the main flow
Post conditions	None
Alternative flows	None

Use Case	ChooseBuildWay
ID	UC5
Brief Description	The user chooses the way to build
Primary Actors	User
Secondary Actors	System
Pre conditions	Two radio buttons are displayed by system
Main flow	1. The user chooses the way to build by clicking one of the buttons 2. The system displays “Start” button
Post conditions	A “Start” button is displayed
Alternative flows	None

Use Case	StartTasks
ID	UC6
Brief Description	The system will start a series of tasks including tagging, building and deployment
Primary Actors	System
Secondary Actors	User
Pre conditions	The user has chosen the correct way to build
Main flow	1. The user clicks “Start” button 2. The system starts tagging task 3. After tagging task, the system starts building task 4. During building task, the system starts deployment task 5. After all the tasks finished, the system displays all the results and the “Refresh” button
Post conditions	None
Alternative flows	None

Use Case	ManualBuild
ID	UC6.1
Brief Description	The user chooses to manually build the component
Primary Actors	System
Secondary Actors	User
Pre conditions	The system display “Continue” button and prompt user to manually build the component
Main flow	1. The alternative flow starts after step 3 of the main flow. 2. The user manully build the component 3. The user clicks “Continue” button 4. The system continues with auto build process.
Post conditions	None
Alternative flows	None

Use Case	Refresh
ID	UC7
Brief Description	The user refreshes the release process
Primary Actors	User
Secondary Actors	System
Pre conditions	A “Refresh” button is displayed

Main flow	1. The user clicks the button 2. Return to the use case 1
Post conditions	A “Start” button is displayed
Alternative flows	None

4.4.1 Release automation module

Release_automation class is the core class in the client side for the reason that it is the class that align all the tasks together and display the progress in correct work flow for users. It is achieved via defining correspondent actions of different widget respectively.

The problem we encountered during the implementation phase was how to display and progress and widgets following the work flow and also to make the application efficient. According to the web interface design, we created various of widgets to help interacting with users, since different widgets are needed in different stage, we added them when needed and remove them when not needed. When we were trying to do this, the code become nested and hard to read because one widget is defined in an class that another widget is defined. Furthermore, it is inefficient to add and remove one widget for several times.

To solve the problem, we analyzed the features of widgets and then found out we could set up them and their responses at first place but make it invisible. Then we set specific widget to visible in sequence based on the work flow to display it and let the user interact with it. After that, we set the widget invisible when it is not needed.

In this module, we implemented several widgets listed as below:

- ListBox releaseEnvMenu: To display release names
- ListBox releaseNameMenu: To display release environments
- TextBox jiraTicket: To enter Jira ticket for users
- RadioButton install: To indicate the way of building
- RadioButton deploy: To indicate the way of building
- Button startButton: To indicate the start of tasks
- TextArea rpmList: To show the Rpm lists built
- Button refresh: To restart the release process

For all widgets, we defined the actions for buttons following the same rule:

```
//set up button widgets
addWidget();
CreateNewClickHandler();
addClickHandler();

//set up other widgets
addWidget();
CreateNewChangeHandler();
addChangeHandler();
```

Both new click handler and new change handler define the actions widgets would response to click or change events. For different widgets, the new Change handler or the new Click handler are different, so we outlined the functionalities in pseudo code:

- releaseEnvMenu

```
for (allEnvironments){
        if (environmentSelected){
             loadRelease(selectedEnvironment, selectedName);
```

```
         }
}
```

- releaseNameMenu

```
for (allNames){
         if (nameSelected){
              loadRelease(selectedEnvironment, selectedName);
         }
}
```

- jiraTicket

```
checkIfValid();
setInstallButtonVisible();
setDeployButtonVisible();
```

- install and deploy

```
if (getValue()){
setBuildWayString();
setStartButtonVisible();
}
```

- startButton

```
getReleaseWithcheckedComponents(release);
if(releaseWithCheckedComponents!=null){
    new TaggingTask(releaseWithCheckedComponents, automationService,
componentTable, Release_automation.this);
    setDeployButtonInvisible();
    setInstallButtonInvisible();
}
```

- Button refresh: To restart the release process

```
setReleaseEnvMenuVisible();
loadRelease(defaultReleaseEnvironment, selectedReleaseName);
```

To implement the functions of the widget, we also defined some methods so that we could call them when needed. There are three main methods to indicate the progress of release which are loadRelease(releaseEnvironment, releaseName), taggingComplete(taggedComponents) and buildComplete(builtComponents, rpmFilenames)

- loadRelease(releaseEnvironment, releaseName) :

```
automationService.getRelease(releaseEnv, releaseName, new AsyncCallback<Release>() {
                      public void onFailure(Throwable caught) {
                      }

                      public void onSuccess(Release result) {
                              refreshButton.setVisible(false);
                              releaseFromWiki = result;
                              populateComponentTable();
                              if (releaseFromWiki.getComponents().size() > 0)
                                      jiraTicket.setVisible(true);
                              if (jiraTicket.getText().length() > 0) {
                                      startButton.setVisible(true);
                              }
                      }
               });
```

By calling getRelease(releaseEnv, releaseName, new AsyncCallback<Release>), the GWT will execute the getRelease method in AutomaionServiceImpl class and if server is successfully invoked it will return asynchronous callback in new AsyncCallback<Release> and in the onSuccess method, we can pass the release result to variable called releasefromWiki and populate component table with this release as well as set the jiraTicket widget visible.

- taggingComplete(taggedComponents):

```
if (!taggedComponents.isEmpty()) {
                        taggedComponentsResult = taggedComponents;
                        new BuildTask(buildWay, taggedComponentsResult,
automationService, componentTable,
                                        Release_automation.this);
                } else if (taggedComponents.isEmpty()) {
                        refreshButton.setVisible(true);
                }
```

The method taggingComplete(taggedComponents) is the implementation of TaggingTaskCallback interface and it returns a list of tagged components. If the taggedComponents list is not empty, we continue the build process by instantiating a build task, otherwise we set the refresh button visible to give users a choice of restart.

- buildComplete(built Components, rpmFilenames)

```
                if (rpmFilenames.size() > 0) {
                        rpmList.setVisible(true);
                        StringBuilder sb = new StringBuilder();
                        for (String filename : rpmFilenames) {
                                sb.append(filename);
                                sb.append(“\n”);
                        }
                        rpmList.setText(sb.toString());
                }
                if (rpmFilenames != null) {
                        deployButton.setVisible(true);
                }
```

Like taggingComplete(taggedComponents), buildComplete(builtComponents, rpmFilenames) implements BuildTaskCallback interface and it returns a list of built components and also a list of rpm file names. If the list of rpm file names is not empty, then we set rpmList widget visible, pass the file name strings to a textArea widget called rpmFilenames and set deployButton to visible.

By setting up the widgets and implementing these methods, we could align the process of tagging, building and deployment and display each process in correct order as well as allow users to interact with it.

Now we will explain how TaggingTask, BuildTask implemented by describing three important elements of tasks in details.

4.4.2 Tasks

As we mentioned before, TaggingTask and BuildTask do actual tagging and building and get callback of results or exceptions from server side. To achieve this, we

To do actual tagging and building, TaggingTask and BuildTask. We will take BuildTask as an example. To achieve the functions, we used LinkedList to store the auto built component. We leveraged asynchronous calls to build components by invoking the buildComponentImpl() method in server side. Moreover, we used timer to refresh the process of getting results and exceptions dynamically.

To explain the work flow of □pmFilena, we outlined pseudo code as below:

```
                        // Add all the tagged components to a linked list queue
                        buildComponentQueue.addAll(taggedComponents);
```

```
                      // Display component table as build task starts

                      componentTable.componentStartbuilding(taggedComponents);
                      // Add auto built component to an arrayList
                      if (!c.isManualBuild()) {
                          autoBuildComponents.addI;
                }
                       // Start timer to get results and exceptions dynamically
                       startServerPoller();
                       //set up continue button for continuing auto build after
component is manually built
                       continueBuildButton.addClickHandler(new ClickHandler() {
                      public void onClick(event) {
                              continueBuildButton.setInvisible
                              retrieveBuildQueue();
                       }
               });
                        // Retrieve component from the LinkedList queue
                        currentComponent=buildComponentQueue.poll();
                        if (autoBuildComponents.contains(currentComponent)){
                           componentTable.
displayAutoBuiltComponent(currentComponent);
                        }else{

componentTable.displayManualBuiltComponent(currentComponent);
                            comtinueBuildButton.setVisible();
                        }
                        // Build component
                      automationService.buildComponent(buildWay,component,
newAsyncCallback<Void>){
                          success(){
                          retrieveBuildQueue();
                          }
                      }
```

To store the tagged components, we chose LinkedList based queue implementation. The logic behind that is that we need a dynamic data structure since we don't know how many components to be stored. A linked list works by creating a collection of objects which carry both data and a reference to the next node. If it is implemented by queue, we could retrieve it in the first-in-first-out order. In the retrieveBuildQueue() method, we could retrieve every component in sequence and check if it is contained in the auto built components list and decide whether we pass it to buildComponent(Component c) or not.

If the component is contained in auto built component lists, in order to do the actual building, we applied the asynchronous remote procedure call. In buildComponent(Component c) method, we invoked the server side by calling buildComponent(buildWay, c, new AsyncCallback<Void>) defined in client side. If the call is made successfully, the buildComponent implementation in server side will do the actual building by instantiating buildComponentImpl() object. After the component is built successfully, we could retrieve the next component in the queue. That is why we call retrieveBuildQueue() method in onSuccess(Void obj).

To get the results or exceptions, we defined method startServerPoller(). We made two asynchronous remote procedure calls to server, one to get results and one to get exceptions.

Here the interesting part is that we leveraged Timer class to get dynamic callbacks and control the time to refresh. By calling method run() when the timer is fired, we can execute the process of getting the results and exceptions. When the amount of results and exceptions added together exceeds the number of components in build queue, all the components were built either manually or automatically, then we can cancel the timer by calling cancel() method. We also scheduled the timer to elapse repeatly every one second by calling scheduleRepeating(1000). By using

Timer, we could get the callback of results and exceptions constantly and repeatedly during the process.

4.4.3 User interface: component table

Besides widgets we defined in Release_automation, the main element to display user interface is ComponentTable. Class ComponentTable extends FlexTable and we could call methods of FlexTable directly. All the functions to display release tables are achieved in class FlexTable, the methods are listed as below:

- populateWithRelease(release);
- componentStartTagged(release);
- componentHasTagged(taggingResult);
- componentTaggingFailed(taggingException);
- componentStartBuild(component);
- displayManualBuildComponent(component);
- displayAutoBuildComponent(component);
- componentHasBuilt(buildResult);
- componentBuildFailed(buildException);

In the methods, we displayed the release table according to different stages. First, we displayed a table with only component, version and deployment environment. Then when tagging task begins, we added another column to indicate the tagging progress.

In order to show the status of each component, we need to have spinner icon indicating is it in the process of tagging, also tick icon indicating it is tagged successfully as well as red cross showing failure. Instead of using small pictures, we used image bundle to create these icons.

Image bundle is a composition of many images into a single image, along with an interface for accessing the individual images from within the composite. We can define an image bundle that contains images we want to use and GWT will automatically create the composite image and provide an implementation of the interface for accessing each individual image. In this way, we could replace round trip to the server for each image with only one to the server for the composite image which greatly speed up our application. After we defined the image bundle, we could use the method setWidget FlexTable inherited from class HTMLTable to set the image for specific row.

Chapter 5 Testing

In this chapter, we will describe the testing of the implemented project including Junit test of sub-component project and the release test for the whole project.

5.1 Junit Test

Testing is a crucial part of our software development cycle. In our project, several sub-component projects were being developed at the same time and integrated into one at last. We need to do unit test first for each component project to check the basic function of code, to document the interfaces exposed in classes and to locate bugs and fix bugs. Once we make sure all the sub-component is working property separately, we group them together and do actual practice and real QA releases to test the whole function.

Junit test is a unit testing framework for Java programming language. In order to generate a Junit test, annotate a method with @org.junit.Test first. Then set up a program structure to call the functions we want to test and compare the results of the call with the expected ones by by calling assertionTrue(). If the test is successful, the method will pass a □pmFile true.

Take the Junit test for a function in Deployer as an example. In Deployer, we defined a method getDeployedComponentVersions(rpmText) to get a map of component name and its corresponding version deployed on host server. To test its function, we wrote a Junit test.

```
//set up expected deployed component versions
               Map<Component, Map<String, String>> expectedDeployedComponentVersions =
new TreeMap<Component, Map<String, String>>();
               Map<String, String> expectedMap1 = new HashMap<String, String>();
               Map<String, String> expectedMap2 = new HashMap<String, String>();
               List<Component> components = new ArrayList<Component>();
               List<String> hostnames = new ArrayList<String>();
               hostnames.add(“ukbvmdevorc001”);
               Component component1 = new Component(“quotes-feed”, “1.4.0.1”, new
Habitat(“App”, hostnames));
               Component component2 = new Component(“staticdata-service”, “1.7.0.2”,
new Habitat(“APP”, hostnames));
               components.add(component1);
               components.add(component2);
               Release release = new Release(“Dominica”, components);
               expectedMap1.put(“ukbvmdevorc001”, “1.4.0.1”);
               expectedMap2.put(“ukbvmdevorc001”, “1.7.0.2”);
               expectedDeployedComponentVersions.put(component1, expectedMap1);
               expectedDeployedComponentVersions.put(component2, expectedMap2);
//do actual function and get actual result
               Map<Component, Map<String, String>> deployedComponentVersions = new
TreeMap<Component, Map<String, String>>();
               ComponentDeployerImpl componentDeployerImpl = new
ComponentDeployerImpl(“release”, “eight8”,
                              “/home/release”);
               deployedComponentVersions =
componentDeployerImpl.getDeployedComponentVersions(release);
                       Assert.assertEquals(expectedDeployedComponentVersions,
deployedComponentVersions);
```

We began by specifying the expected result of method giving a rpm text as argument . Then we instantiate an object of Deployer class and call getPackageVersions(text). Then we use assertTrue and assertEquals methods in Assert class to compare the expected results and actual results one by one. If they are all successful, we pass the Junit test. If one fails, Junit test fails too.

All sub-components were tested □pmFilenam using Junit framework to make sure its' functions are achieved and bugs are all fixed. After that, we integrated them all together to one project and test it as a whole by doing QA release.

5.2 Release Test

To test all the functions aligned together, we did practice QA release at first. Since it is a test we defined by ourselves, we set up a few criteria for a test to pass. The criterias are listed as below:

- The status and information of each component during each process should be showed with correct icons: If a component is waiting to be tagged, the tagged column will display a spinner; if it is successfully tagged, a tick is showed.; if it failed tagging, a red cross will be displayed and if you hover it, the error message will be displayed.
- Every widget should execute the defined operation when triggered: Widgets including buttons, menu bars will do the specified actions with the correct values
- The interface is intuitive and easy to use: The interface layout should be simple and matches with web page conventions. Users will have no problems interacting with it.

Since the project is associated with source control and the release software

EVB is managed by Maven, a subversion control tool. We can not do as many tests as the other project because doing real release tests for all the components might mess up the repository and cause some problems.

The way we solved it is to create a test release page in confluence page with several newly created test projects. We stored the test projects in repository and treated them as components that makes up a software project. Then we performed tests with this test project for several times to test different scenarios like choosing all components or only several components, choosing building way of deploying or installing. During the first trial, we encountered problem of components not getting the additional attributes of subversion root and parent project which directly leads to tagging failure. The reason is that in Scraper we treated components in two tables differently and in web interface we only passed the components with basic attributes to tagger. We resolved the problem by making changes and update functionalities in Scraper. Then we ran the test again and it was successful.

After doing several QA practice release tests, we felt confident to perform QA release for the actual Evaluated Bonds project. There was no issue with the functions of Release Builder tool and the release was successfully performed.

Chapter 6 Conclusions

This chapter includes a brief conclusion of our project including a summary of what we have done as a team and individually. After that, we will assess build automation tool and suggest future improvement of our project.

6.1 Summary

To solve the problems existing in the software releases of the EVB team at Markit Group Ltd, we have come up with the idea of Release Builder. Release Builder is aimed to design and implement a tool to minimize human errors and increase productivity by automating code build and deployment.

The design of this project falls into several sub-component projects based on the requirements. Each sub-component specifically encapsulates an element in the release, so we designed each project according to the functionalities it performs.

Then the project is implemented using Java programming language and frameworks such as GWT to achieve the functions of web application. Tests were performed to ensure the correct operations of the application including Junit tests for sub-components and release tests for the whole project. At last, we presented some results and screenshots to illustrate the usage of Release Build tool in doing a release.

6.2 Critical Assessment of Release Build Tool

Overall, we believe this project has been successful for two reasons:

- The tool has met all the requirements we gathered and achieved all the objectives.
- Its deliverables of design, implementation, tests and final products are on time.

The benefits of Release Build Tool are proved during the first trial releases we performed. The automation functionality we implemented greatly increases productivity for release managers and reduces human errors. Before, release managers had to sit around the desk typing command line by line to do a release and it usually takes half a day, sometimes even a whole day. During the release process, they can not do anything but wait until it is finished. With this tool, release managers could compress more activity into available time. However, the software is still evolving and functions were updated and added during each trial. We believe with further release iterations, we will get more benefits.

Due to the large amount of work in a fairly short time, there are some possibilities to extend and enhance our project. In a broad perspective, we could develop the Release Builder tool into a more standardized project that applies to different projects and different releases. Currently, this tool is customized to the release of EVB team's projects. It is designed and implemented according to the current release procedures in our own team. Since Markit Group Ltd has a variety of products that releases from time to time, if other teams could use this tool also, it would increase productivity of the company.

In terms of functionalities of this tool, there are room for improvement. For Scraper, functions of uploading and downloading release manifest XML file of a release could be performed in case Confluence goes down. For Tagger, SNAPSHOT dependency check could be performed to ensure the safety of tagging. For web interface, we could add a log screen to show previously executed builds with name of the person who triggered them. We could utilize more CSS and HTML features in GWT framework to make the layout more user friendly and more intuitive. Furthermore, more advanced option could be added such as emailing results to the whole team when a release is done.

6.3 Future Work

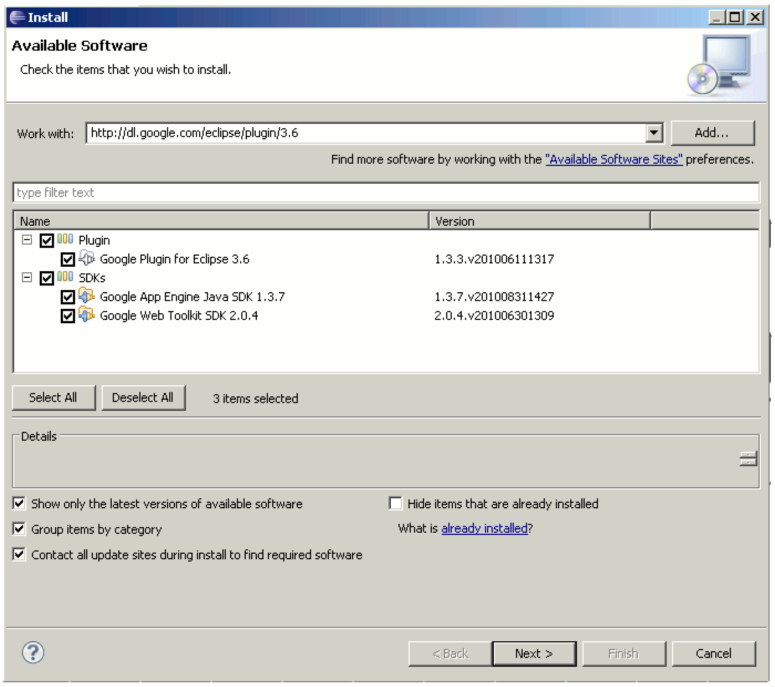
As we discussed in the critical assessment section, some elements could be extended and improved for our project. We listed some of the functions or areas that could be implemented in the future:

- Standard the project and makes it more flexible
- Upload and download of release manifest XML file in Scraper
- SNAPSHOT dependency check in Tagger
- Log screen showing previously executed builds in Web interface
- More CSS and HTML style applied in Web interface
- Advanced functions such as email notification added

Appendix A: System Manual

The application is developed in the Java programming language using Eclipse Helios version. The user interface of this application is developed using GWT framework. So before you run the application, system should be set up first as below:

1. Install the Java SDK
   You could download and install Sun Java Standard Edition SDK from http://www.oracle.com/technetwork/java/javase/downloads/index.html
2. Install Eclipse Helios or other Java IDE
3. Install the Google Plugin and Google Web Toolkit to create and develop GWT applications
   - Select the **Help** Menu, choose **Install New Software**
     In the **work with** box, enter: http://dl.google.com/eclipse/plugin/3.6 (If you are using Helios version of Eclipse. If you are using Galileo, enter http://dl.google.com/eclipse/plugin/3.5)
   - Click "Plugin" and "SDKs" to install the Plugin as well as GWT App Engine Java SDK and Google Web Toolkit SDK in the Eclipse plugin directory(Screenshot A)
   - Click **Next** and accept the terms of service
   - Restart Eclipse and the plugin is installed.



Available Software

Check the items that you wish to install.

Work with: http://dl.google.com/eclipse/plugin/3.6

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
⊟ ☑ Plugin	
☑ Google Plugin for Eclipse 3.6	1.3.3.v201006111317
⊟ ☑ SDKs	
☑ Google App Engine Java SDK 1.3.7	1.3.7.v201008311427
☑ Google Web Toolkit SDK 2.0.4	2.0.4.v201006301309

3 items selected

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☑ Contact all update sites during install to find required software

Screenshot A

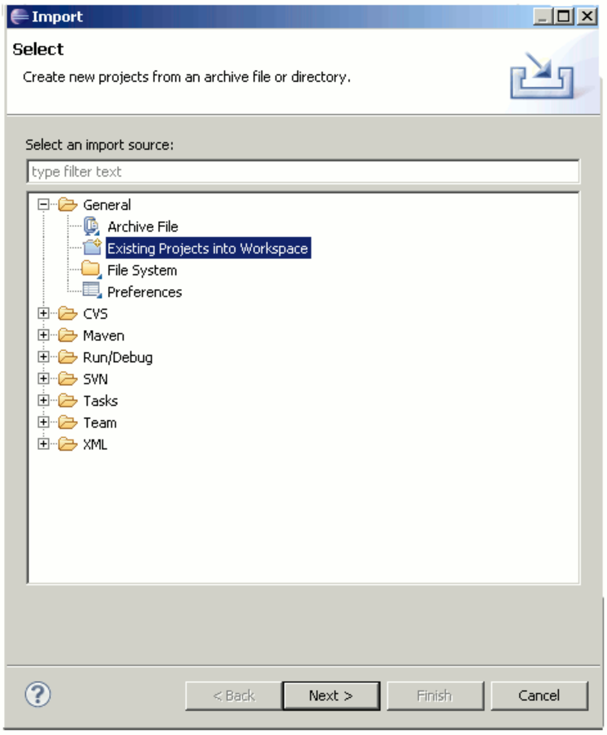
After all the steps have been done, users will be able to run the project in Eclipse as web application.

Appendix B: User Manual

This user manual will illustrate to users how to use the application.

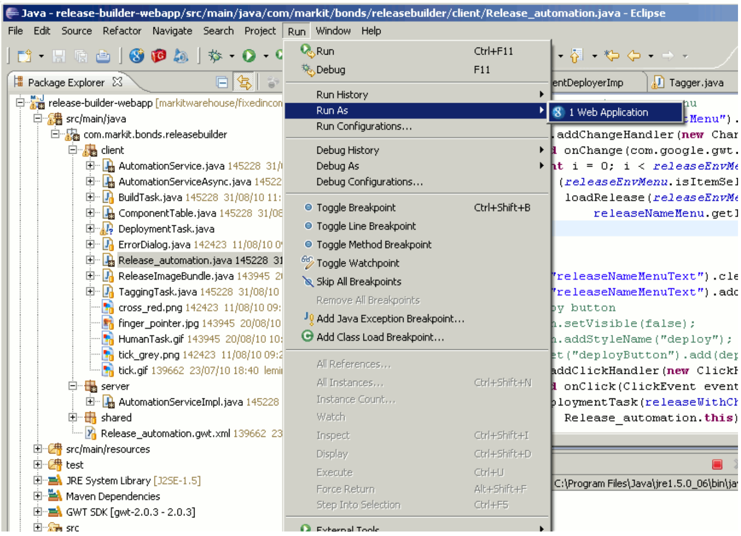
1. Importing the project into Eclipse

- In Eclipse, select “File” from the menu bar and “Import” menu item
- Choose “Existing Projects into Workspace”
- Enter the directory of the project file
- Click **Finish**, the Release Builder project is in the Eclipse workspace

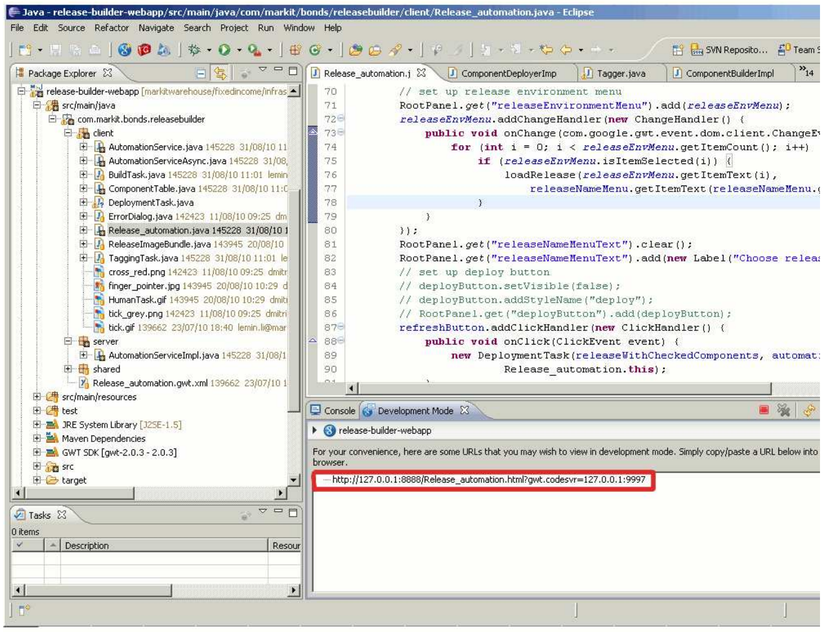


2. Running the system

- In Eclipse, select “Run” from the menu bar
- Select “Run as”
- On the pop up window, click on the “Web Application” and click “OK”



- Copy the Url in the Development Mode and paste it in your web browser, the application will be running.



Java - release-builder-webapp/src/main/java/com/markit/bonds/releasebuilder/client/Release_automation.java - Eclipse

```
// set up release environment menu
RootPanel.get("releaseEnvironmentMenu").add(releaseEnvMenu);
releaseEnvMenu.addChangeHandler(new ChangeHandler() {
    public void onChange(com.google.gwt.event.dom.client.ChangeE
        for (int i = 0; i < releaseEnvMenu.getItemCount(); i++)
            if (releaseEnvMenu.isItemSelected(i)) {
                loadRelease(releaseEnvMenu.getItemText(i),
                    releaseNameMenu.getItemText(releaseNameMenu.
            }
    }
});
RootPanel.get("releaseNameMenuText").clear();
RootPanel.get("releaseNameMenuText").add(new Label("Choose releas
// set up deploy button
// deployButton.setVisible(false);
// deployButton.addStyleName("deploy");
// RootPanel.get("deployButton").add(deployButton);
refreshButton.addClickHandler(new ClickHandler() {
    public void onClick(ClickEvent event) {
        new DeploymentTask(releaseWithCheckedComponents, automat
                Release_automation.this);
```

For your convenience, here are some URLs that you may wish to view in development mode. Simply copy/paste a URL below into browser.

http://127.0.0.1:8888/Release_automation.html?gwt.codesvr=127.0.0.1:9997

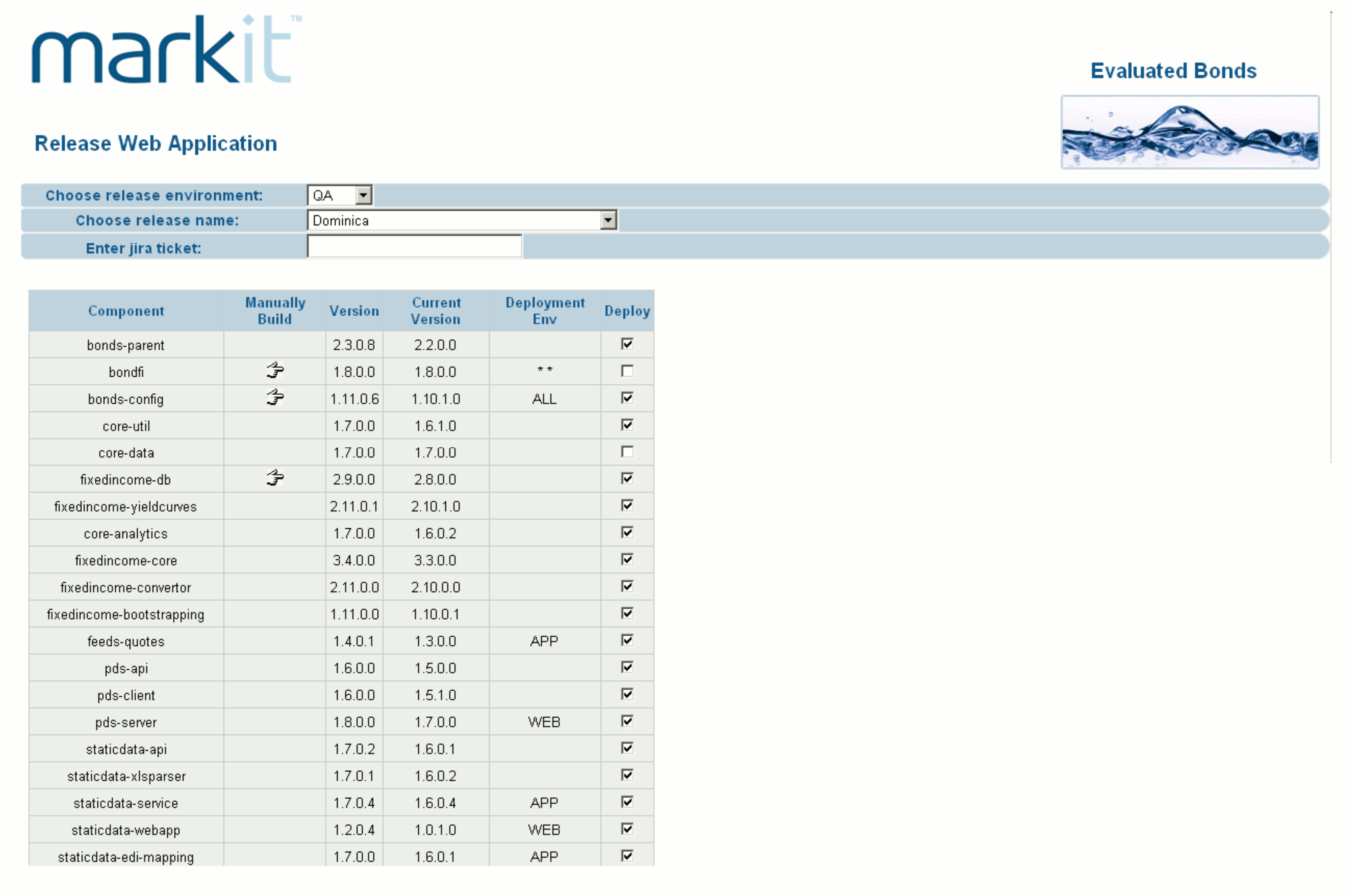
3. Interacting with the web interface

After the second step, the application will start running in the web browser. Users will be able to perform a release by interacting with the interface.

- Application loads the table of current release and default release environment
- User can alter release environment or release name, the table will be reloaded
- After that, user must enter the assigned Jira ticket for the release in the “Enter Jira ticket” text area to carry on
- When Jira ticket is entered, two radio buttons will appear below the table, user can choose the way to build, either install or deploy
- After user selects all the things, “Start” button will appear. The user will able to click “Start” button to start the release process.
- The system will do tagging first and it will display component’s status one by one using different icons. “grey tick” means the component is already tagged, “green tick” means it is tagged successfully while “red cross” means the component failed tagging. If the user hovers on the “red cross”, the error message
- After tagging completes, the system will automatically start building task. In the building task, some of the components are manually built. When it comes to the component that should be manually built, a “Continue” button will be displayed below the table. When the user finishes manual building,

Appendix C: Screenshots

- **Load release**



Release Web Application

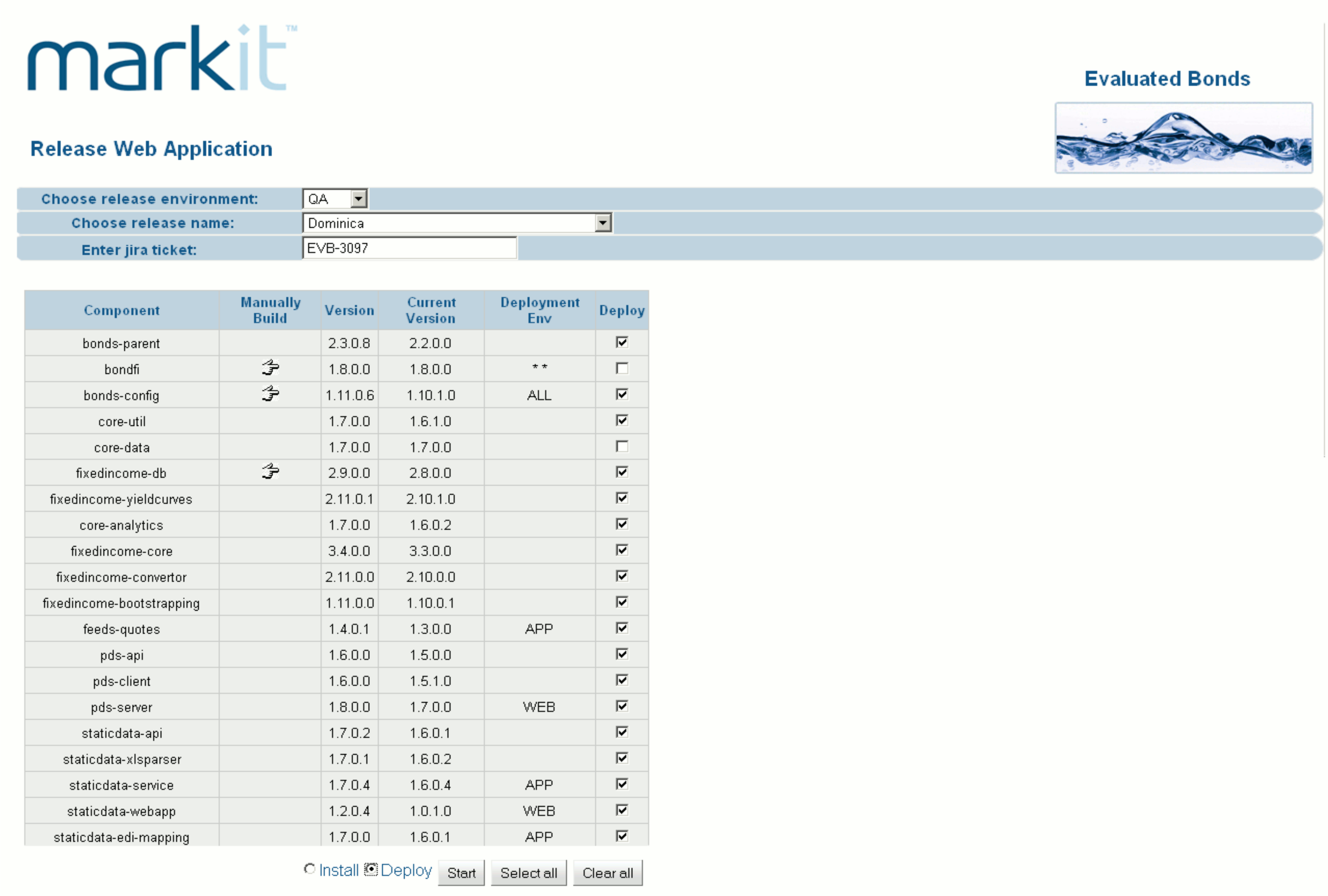
Choose release environment: QA

Choose release name: Dominica

Enter jira ticket:

Component	Manually Build	Version	Current Version	Deployment Env	Deploy
bonds-parent		2.3.0.8	2.2.0.0		☑
bondfi	☛	1.8.0.0	1.8.0.0	* *	□
bonds-config	☛	1.11.0.6	1.10.1.0	ALL	☑
core-util		1.7.0.0	1.6.1.0		☑
core-data		1.7.0.0	1.7.0.0		□
fixedincome-db	☛	2.9.0.0	2.8.0.0		☑
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		☑
core-analytics		1.7.0.0	1.6.0.2		☑
fixedincome-core		3.4.0.0	3.3.0.0		☑
fixedincome-convertor		2.11.0.0	2.10.0.0		☑
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		☑
feeds-quotes		1.4.0.1	1.3.0.0	APP	☑
pds-api		1.6.0.0	1.5.0.0		☑
pds-client		1.6.0.0	1.5.1.0		☑
pds-server		1.8.0.0	1.7.0.0	WEB	☑
staticdata-api		1.7.0.2	1.6.0.1		☑
staticdata-xlsparser		1.7.0.1	1.6.0.2		☑
staticdata-service		1.7.0.4	1.6.0.4	APP	☑
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	☑
staticdata-edi-mapping		1.7.0.0	1.6.0.1	APP	☑

- **Enter Jira ticket and choose build way**



Release Web Application

Choose release environment: QA

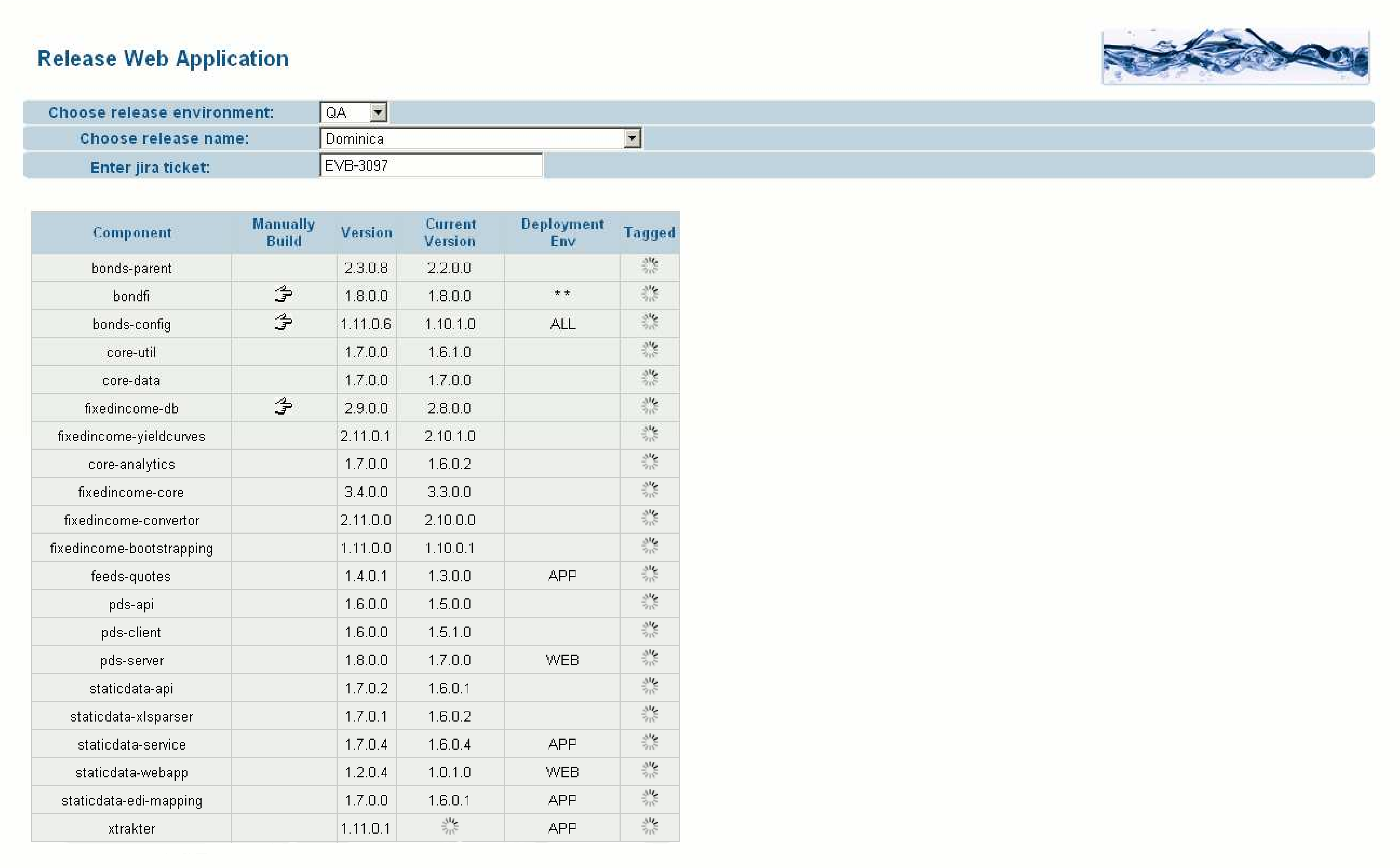
Choose release name: Dominica

Enter jira ticket: EVB-3097

Component	Manually Build	Version	Current Version	Deployment Env	Deploy
bonds-parent		2.3.0.8	2.2.0.0		☑
bondfi	☞	1.8.0.0	1.8.0.0	* *	☐
bonds-config	☞	1.11.0.6	1.10.1.0	ALL	☑
core-util		1.7.0.0	1.6.1.0		☑
core-data		1.7.0.0	1.7.0.0		☐
fixedincome-db	☞	2.9.0.0	2.8.0.0		☑
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		☑
core-analytics		1.7.0.0	1.6.0.2		☑
fixedincome-core		3.4.0.0	3.3.0.0		☑
fixedincome-convertor		2.11.0.0	2.10.0.0		☑
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		☑
feeds-quotes		1.4.0.1	1.3.0.0	APP	☑
pds-api		1.6.0.0	1.5.0.0		☑
pds-client		1.6.0.0	1.5.1.0		☑
pds-server		1.8.0.0	1.7.0.0	WEB	☑
staticdata-api		1.7.0.2	1.6.0.1		☑
staticdata-xlsparser		1.7.0.1	1.6.0.2		☑
staticdata-service		1.7.0.4	1.6.0.4	APP	☑
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	☑
staticdata-edi-mapping		1.7.0.0	1.6.0.1	APP	☑

○ Install ● Deploy

- **Tagging start**

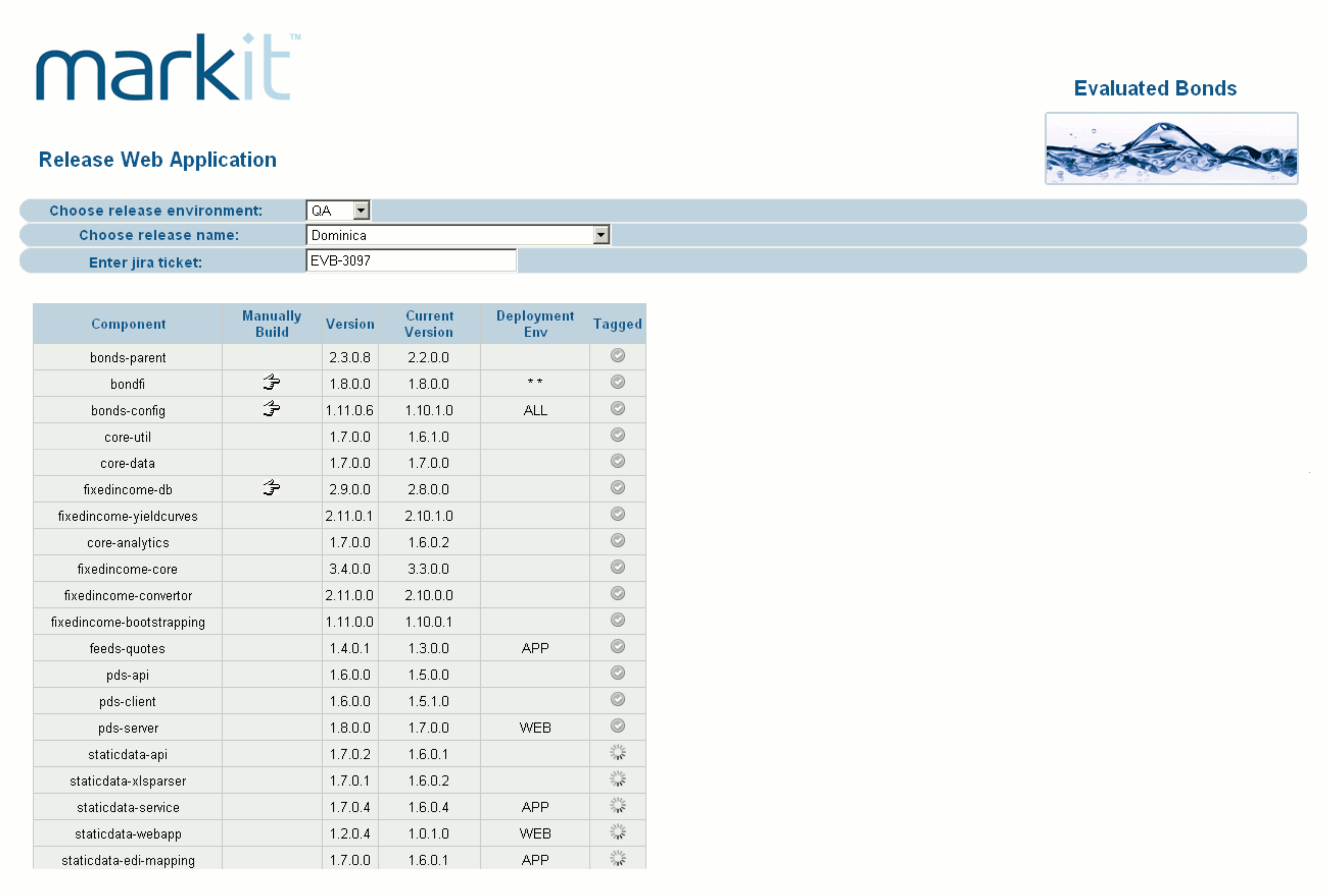


Release Web Application

Choose release environment:	QA
Choose release name:	Dominica
Enter jira ticket:	EVB-3097

Component	Manually Build	Version	Current Version	Deployment Env	Tagged
bonds-parent		2.3.0.8	2.2.0.0		⟳
bondfi	☞	1.8.0.0	1.8.0.0	* *	⟳
bonds-config	☞	1.11.0.6	1.10.1.0	ALL	⟳
core-util		1.7.0.0	1.6.1.0		⟳
core-data		1.7.0.0	1.7.0.0		⟳
fixedincome-db	☞	2.9.0.0	2.8.0.0		⟳
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		⟳
core-analytics		1.7.0.0	1.6.0.2		⟳
fixedincome-core		3.4.0.0	3.3.0.0		⟳
fixedincome-convertor		2.11.0.0	2.10.0.0		⟳
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		⟳
feeds-quotes		1.4.0.1	1.3.0.0	APP	⟳
pds-api		1.6.0.0	1.5.0.0		⟳
pds-client		1.6.0.0	1.5.1.0		⟳
pds-server		1.8.0.0	1.7.0.0	WEB	⟳
staticdata-api		1.7.0.2	1.6.0.1		⟳
staticdata-xlsparser		1.7.0.1	1.6.0.2		⟳
staticdata-service		1.7.0.4	1.6.0.4	APP	⟳
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	⟳
staticdata-edi-mapping		1.7.0.0	1.6.0.1	APP	⟳
xtrakter		1.11.0.1	⟳	APP	⟳

- **Tagging Process**



markit™

Evaluated Bonds

Release Web Application

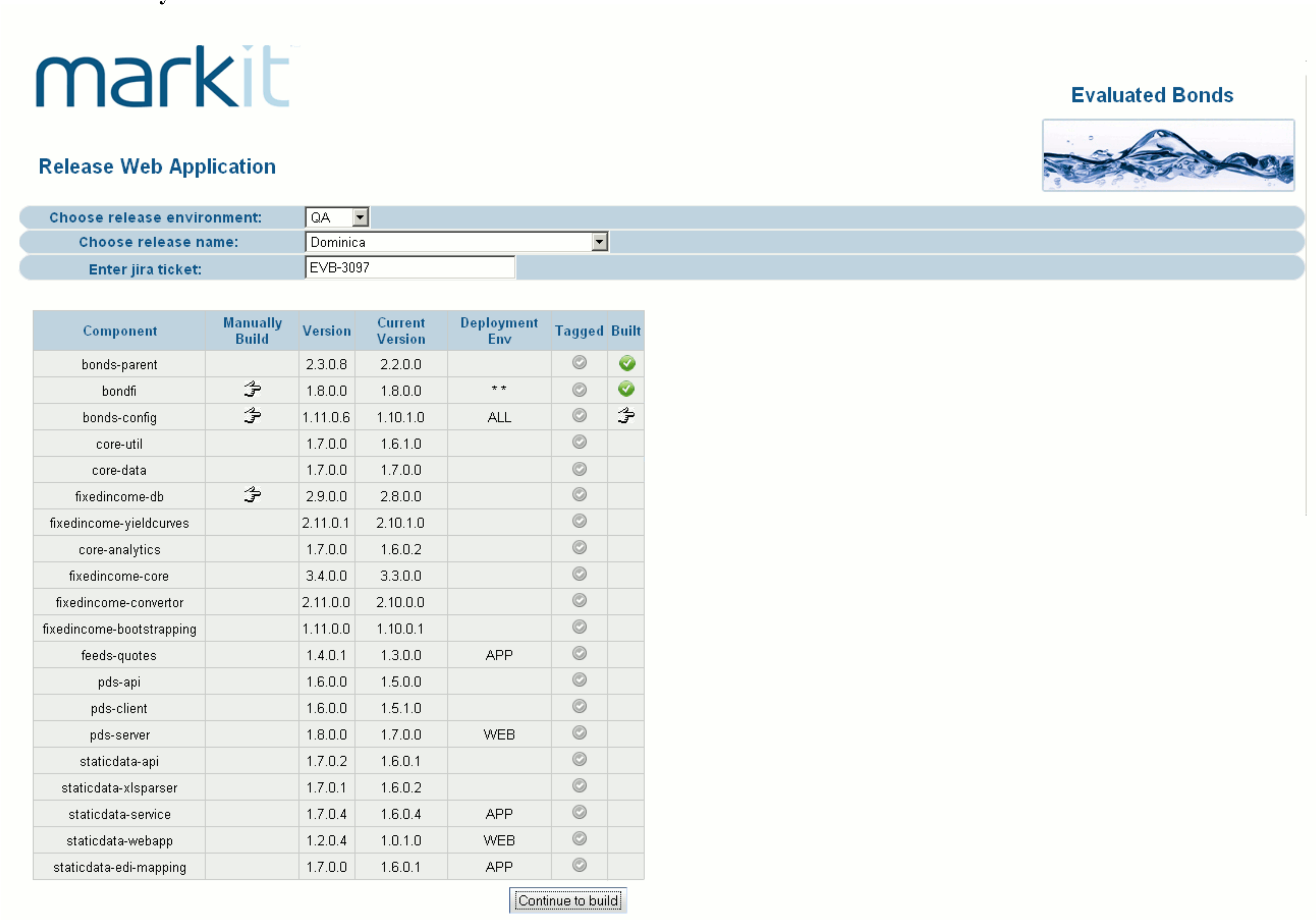
Choose release environment: QA

Choose release name: Dominica

Enter jira ticket: EVB-3097

Component	Manually Build	Version	Current Version	Deployment Env	Tagged
bonds-parent		2.3.0.8	2.2.0.0		✓
bondfi	☞	1.8.0.0	1.8.0.0	* *	✓
bonds-config	☞	1.11.0.6	1.10.1.0	ALL	✓
core-util		1.7.0.0	1.6.1.0		✓
core-data		1.7.0.0	1.7.0.0		✓
fixedincome-db	☞	2.9.0.0	2.8.0.0		✓
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		✓
core-analytics		1.7.0.0	1.6.0.2		✓
fixedincome-core		3.4.0.0	3.3.0.0		✓
fixedincome-convertor		2.11.0.0	2.10.0.0		✓
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		✓
feeds-quotes		1.4.0.1	1.3.0.0	APP	✓
pds-api		1.6.0.0	1.5.0.0		✓
pds-client		1.6.0.0	1.5.1.0		✓
pds-server		1.8.0.0	1.7.0.0	WEB	✓
staticdata-api		1.7.0.2	1.6.0.1		⟳
staticdata-xlsparser		1.7.0.1	1.6.0.2		⟳
staticdata-service		1.7.0.4	1.6.0.4	APP	⟳
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	⟳
staticdata-edi-mapping		1.7.0.0	1.6.0.1	APP	⟳

- **Manually Build**

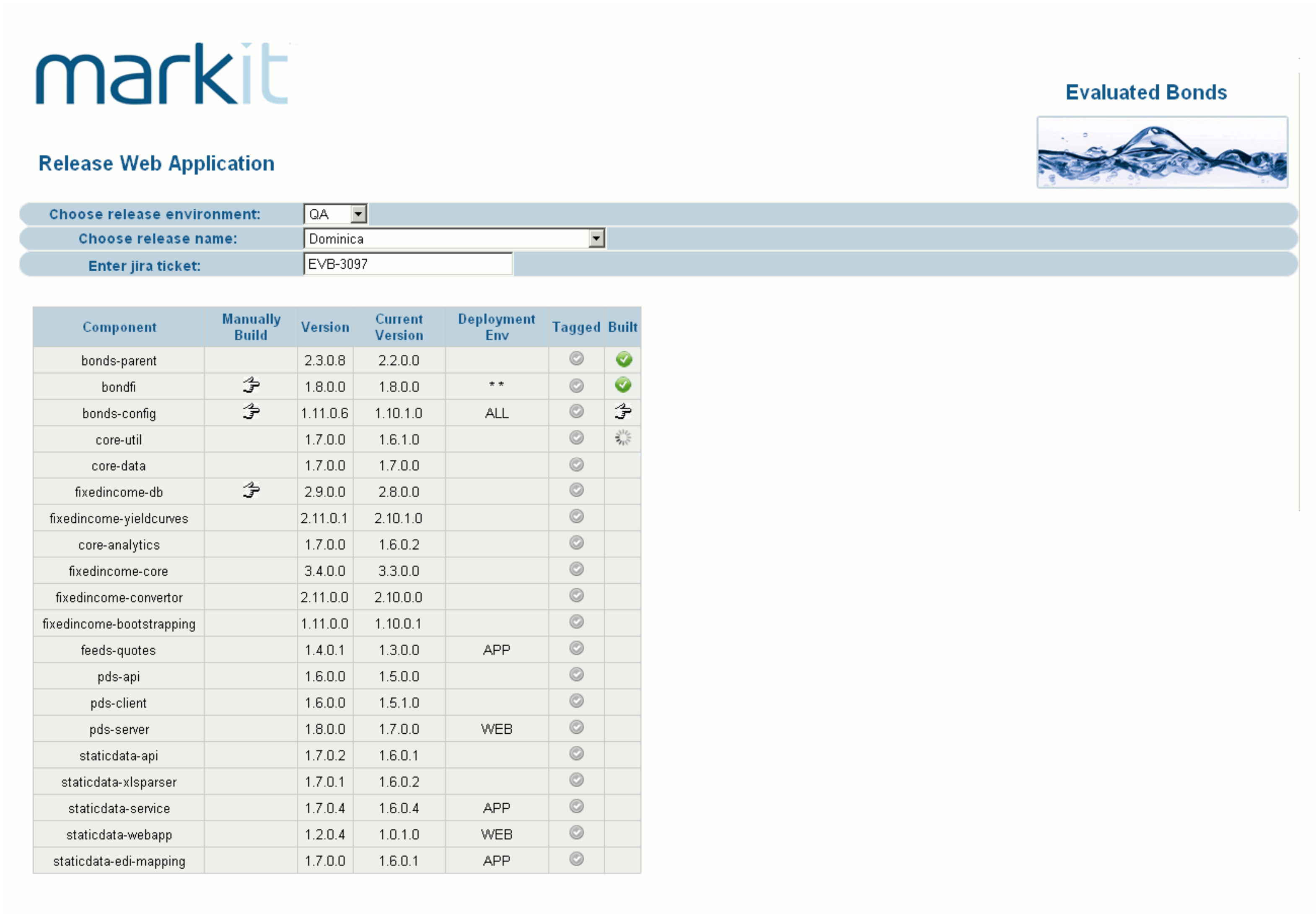


Release Web Application

Choose release environment:	QA
Choose release name:	Dominica
Enter jira ticket:	EVB-3097

Component	Manually Build	Version	Current Version	Deployment Env	Tagged	Built
bonds-parent		2.3.0.8	2.2.0.0		○	✓
bondfi	☞	1.8.0.0	1.8.0.0	* *	○	✓
bonds-config	☞	1.11.0.6	1.10.1.0	ALL	○	☞
core-util		1.7.0.0	1.6.1.0		○	
core-data		1.7.0.0	1.7.0.0		○	
fixedincome-db	☞	2.9.0.0	2.8.0.0		○	
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		○	
core-analytics		1.7.0.0	1.6.0.2		○	
fixedincome-core		3.4.0.0	3.3.0.0		○	
fixedincome-convertor		2.11.0.0	2.10.0.0		○	
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		○	
feeds-quotes		1.4.0.1	1.3.0.0	APP	○	
pds-api		1.6.0.0	1.5.0.0		○	
pds-client		1.6.0.0	1.5.1.0		○	
pds-server		1.8.0.0	1.7.0.0	WEB	○	
staticdata-api		1.7.0.2	1.6.0.1		○	
staticdata-xlsparser		1.7.0.1	1.6.0.2		○	
staticdata-service		1.7.0.4	1.6.0.4	APP	○	
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	○	
staticdata-edi-mapping		1.7.0.0	1.6.0.1	APP	○	

- **Build Process**



Release Web Application

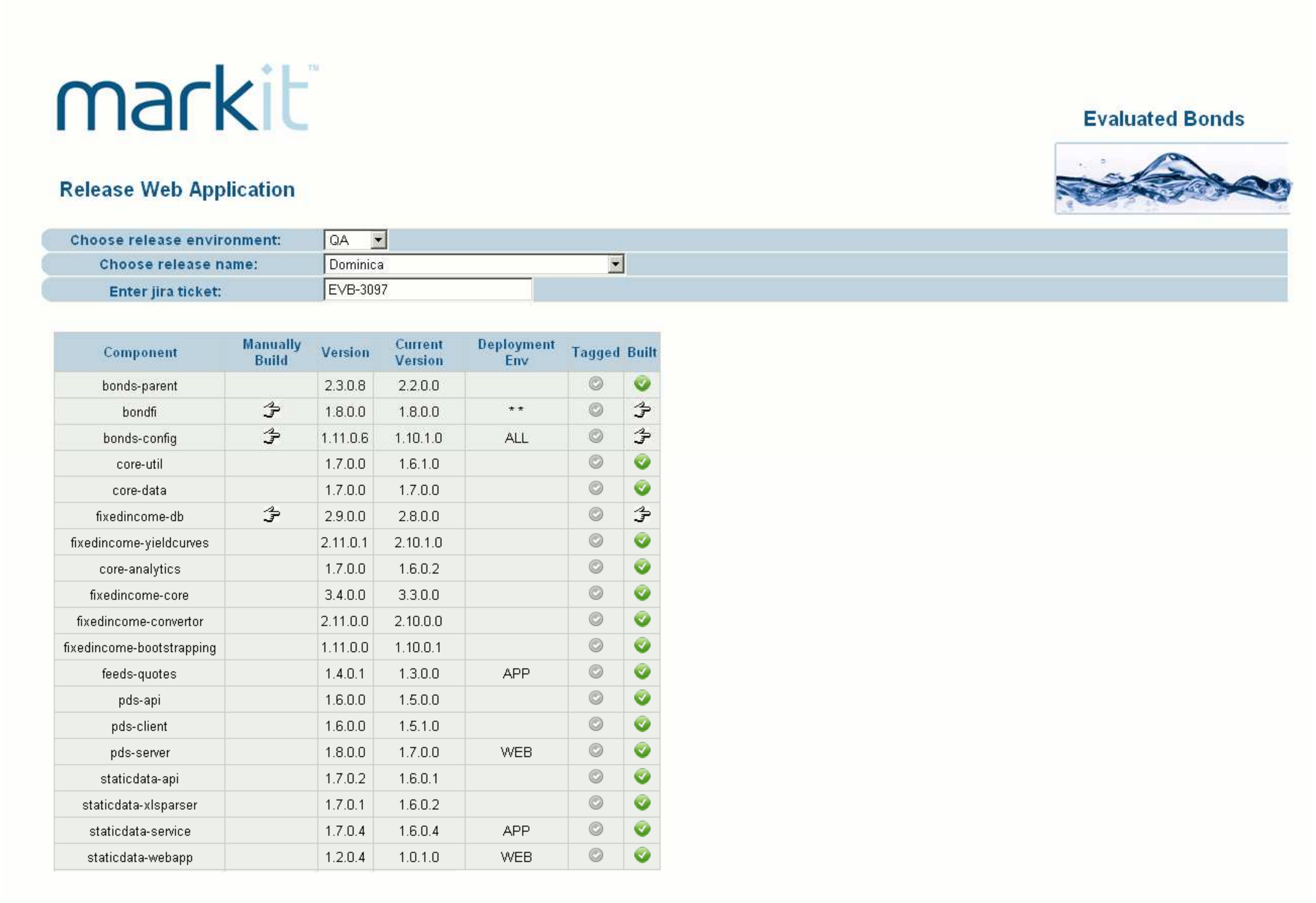
Choose release environment: QA

Choose release name: Dominica

Enter jira ticket: EVB-3097

Component	Manually Build	Version	Current Version	Deployment Env	Tagged	Built
bonds-parent		2.3.0.8	2.2.0.0		○	✓
bondfi	☞	1.8.0.0	1.8.0.0	* *	○	✓
bonds-config	☞	1.11.0.6	1.10.1.0	ALL	○	☞
core-util		1.7.0.0	1.6.1.0		○	⟳
core-data		1.7.0.0	1.7.0.0		○	
fixedincome-db	☞	2.9.0.0	2.8.0.0		○	
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		○	
core-analytics		1.7.0.0	1.6.0.2		○	
fixedincome-core		3.4.0.0	3.3.0.0		○	
fixedincome-convertor		2.11.0.0	2.10.0.0		○	
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		○	
feeds-quotes		1.4.0.1	1.3.0.0	APP	○	
pds-api		1.6.0.0	1.5.0.0		○	
pds-client		1.6.0.0	1.5.1.0		○	
pds-server		1.8.0.0	1.7.0.0	WEB	○	
staticdata-api		1.7.0.2	1.6.0.1		○	
staticdata-xlsparser		1.7.0.1	1.6.0.2		○	
staticdata-service		1.7.0.4	1.6.0.4	APP	○	
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	○	
staticdata-edi-mapping		1.7.0.0	1.6.0.1	APP	○	

- **Build Successful**



Release Web Application

Choose release environment: QA

Choose release name: Dominica

Enter jira ticket: EVB-3097

Component	Manually Build	Version	Current Version	Deployment Env	Tagged	Built
bonds-parent		2.3.0.8	2.2.0.0		○	✓
bondfi	☞	1.8.0.0	1.8.0.0	* *	○	☞
bonds-config	☞	1.11.0.6	1.10.1.0	ALL	○	☞
core-util		1.7.0.0	1.6.1.0		○	✓
core-data		1.7.0.0	1.7.0.0		○	✓
fixedincome-db	☞	2.9.0.0	2.8.0.0		○	☞
fixedincome-yieldcurves		2.11.0.1	2.10.1.0		○	✓
core-analytics		1.7.0.0	1.6.0.2		○	✓
fixedincome-core		3.4.0.0	3.3.0.0		○	✓
fixedincome-convertor		2.11.0.0	2.10.0.0		○	✓
fixedincome-bootstrapping		1.11.0.0	1.10.0.1		○	✓
feeds-quotes		1.4.0.1	1.3.0.0	APP	○	✓
pds-api		1.6.0.0	1.5.0.0		○	✓
pds-client		1.6.0.0	1.5.1.0		○	✓
pds-server		1.8.0.0	1.7.0.0	WEB	○	✓
staticdata-api		1.7.0.2	1.6.0.1		○	✓
staticdata-xlsparser		1.7.0.1	1.6.0.2		○	✓
staticdata-service		1.7.0.4	1.6.0.4	APP	○	✓
staticdata-webapp		1.2.0.4	1.0.1.0	WEB	○	✓

Appendix D: Code Listing

Since the big project comprises of five sub-component projects, there are large amount of code which can not be all displayed here. I will list classes which performs important functions in each of the sub-component project. However, all other codes can be found in the CD enclosed.

- **Scraper:**

ConfluenceScraperImpl class

```
package com.markit.bonds.releasebuilder.confluencescraper;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import com.markit.bonds.coreutil.io.XmlMarshalling;
import com.markit.bonds.releasebuilder.api.ConfluenceScraper;
import com.markit.bonds.releasebuilder.api.model.Component;
import com.markit.bonds.releasebuilder.api.model.Environment;
import com.markit.bonds.releasebuilder.api.model.Habitat;
import com.markit.bonds.releasebuilder.api.model.Release;

public class ConfluenceScraperImpl implements ConfluenceScraper {
    private static final String URL =
"https://confluence.markit.com/display/EVB/Releases?os_authType=basic";
    private static final Integer allComponentsTable = 0;
    private static final Integer autoDeployComponentsTable = 4;
    private final Map<String, Component> compMap;

    /**
     * Get the Confluence Release page document, parse it, looking for the table of
components.
     *
     * @param args
     */
    public static void main(String[] args) {
            ConfluenceScraperImpl confluenceScraper = new ConfluenceScraperImpl();
            for (Component c : confluenceScraper.getRelease("QA",
"Dominica").getComponents()) {
                    System.out.println(c.getName() + c.getHabitat().getHostnames());
            }
    }

    public ConfluenceScraperImpl() {
            compMap = new LinkedHashMap<String, Component>();
    }

    /**
     * Method to get XML file of the release giving a release names
     */
    public String getDomFile(String releaseName, String releaseEnv) {
            try {
                    String doc = XmlMarshalling.marshall(getRelease(releaseName,
releaseEnv));
                    System.out.println(doc);
                    return doc;
            } catch (Exception e) {
                    e.printStackTrace();
            }
            return null;
    }

    /**
```

```
     * Method to get current release name and filter it to only name
     */
    public String getCurrentReleaseName() {
           List<String> allNames = getReleaseNames();
           String currentReleaseName = "";
           for (int I = 0; I < allNames.size(); i++) {
                  if (allNames.get(i).contains("current")) {
                          for (char inputChar :
allNames.get(i).replaceAll("(current)", " ").toCharArray()) {
                                 if (inputChar == '/' ||
Character.isDigit(inputChar) || inputChar == '('
                                                || inputChar == ')' || inputChar
== '?') {
                                         continue;
                                 }
                                 currentReleaseName += inputChar;
                          }
                          return currentReleaseName;
                  }
           }
           return null;
    }

    /**
     * Method to get all release names
     */
    public List<String> getReleaseNames() {
           try {
                  Document doc = ScraperUtil.getDocument(URL);
                  NodeList nodeList = doc.getElementsByTagName("tr");
                  List<String> text = new ArrayList<String>();
                  NodeList tableRow = nodeList.item(0).getChildNodes();
                  int columnIndex = 0;
                  for (int I = 7; I < tableRow.getLength(); I += 2) {
                          Node node = tableRow.item(i);
                          if (node.getNodeType() == Node.ELEMENT_NODE
                                          && ((Element)
node).getTagName().equalsIgnoreCase("th")) {
                                 // We've found a th tag, so check all the
children to see if
                                 // they contain the column text that we're
searching for
                                 columnIndex++;
                                 String name =
ScraperUtil.extractTextFromChildren(node);
                                 text.add((I – 7) / 2, ScraperUtil.clean(name));
                          }
                  }
                  return text;
           } catch (Exception e) {
                  e.printStackTrace();
           }
           return null;
    }

    /**
     *
     */
    public Release getRelease(String releaseEnv, String releaseName) {
           try {
                  Document doc = ScraperUtil.getDocument(URL);
                  Map<String, Component> compMap =
getComponentsMapFromFirstTable(doc, releaseName, releaseEnv);
                  scrapeComponents(doc, releaseName, releaseEnv, compMap);
                  Release release = new Release(releaseName, getComponentList());
                  return release;
           } catch (Exception e) {
                  e.printStackTrace();
           }
           return null;
    }

    public List<Component> getComponentList() {
           List<Component> components = new ArrayList<Component>();
           Set<Entry<String, Component>> compSet = compMap.entrySet();
           for (Entry<String, Component> entry : compSet) {
                  components.add(entry.getValue());
           }
           return components;
    }

    /**
     * Method to get a list of all the components from the first table of
confluence page
     *
     * @param doc
     * @param componentColumnNumber
     * @param releaseColumnNumber
     * @param envColumnNumber
     * @return
     */
    private Map<String, Component> getComponentsMapFromFirstTable(Document doc,
String releaseName,
                   String releaseEnv) {
           int releaseColNum = ScraperUtil.getColumnNumber(doc,
allComponentsTable,
                   releaseName.replaceAll(“\n\r”, “ “).trim());
           int componentColNum = ScraperUtil.getColumnNumber(doc,
allComponentsTable, “component”);
           int envColNum = ScraperUtil.getColumnNumber(doc, allComponentsTable,
“deployed”);
           NodeList tableNodeList = doc.getElementsByTagName(“table”);
           // We’re only interested in the first table on the page (release
components)
           Node releaseTable = tableNodeList.item(allComponentsTable);
           NodeList releaseNodeList = ((Element)
releaseTable).getElementsByTagName(“tr”);
           String componentName = null;
           String version = null;
           String envName = null;
           Environment environment = null;
           // Create new object components to store component name and version in
each row of table
           // List<Component> components = new ArrayList<Component>();
           for (int I = 1; I < releaseNodeList.getLength(); i++) {
                   List<String> habitatHostnames = new ArrayList<String>();
                   componentName = ScraperUtil.getCellText(doc, allComponentsTable,
componentColNum, i).trim();
                   version = ScraperUtil.getCellText(doc, allComponentsTable,
releaseColNum, i).trim();
                   envName = ScraperUtil.trimEnvName(ScraperUtil.getCellText(doc,
allComponentsTable, envColNum, i))
                                   .replaceAll(“DB”, “”).replace(“WINDOWS”, “”);
                   if (envName.contains(“WEB”) || envName.contains(“APP”)) {
                          environment = getEnvironment(releaseEnv, envName);
                          for (Habitat habitat : environment.getHabitats()) {
                                 habitatHostnames = habitat.getHostnames();
                                 // System.out.println(habitatHostnames);
                          }
                   }
                   Habitat habitat = new Habitat(envName, habitatHostnames);
                   Component component = new Component(componentName, version,
habitat);
                   if (ScraperUtil.clean(component.getVersion()) == null) {
                          // Skip this component as it doesn’t have a version
number on the Releases page
                          continue;
                   }
                   compMap.put(componentName, component);
           }
           return compMap;
    }

    private Environment getEnvironment(String releaseEnv, String habitatName) {
           Environment environment = ServiceScraper.getEnvironment(releaseEnv,
habitatName);
           return environment;
    }

    private void scrapeComponents(Document doc, String releaseName, String
releaseEnv,
                   Map<String, Component> compMap) {
           int componentColNum = ScraperUtil.getColumnNumber(doc,
autoDeployComponentsTable, “component”);
```

```
           int svnColNum = ScraperUtil.getColumnNumber(doc,
autoDeployComponentsTable, “svn”);
           int parentColNum = ScraperUtil.getColumnNumber(doc,
autoDeployComponentsTable, “parent”);
           int versionPropColNum = ScraperUtil.getColumnNumber(doc,
autoDeployComponentsTable, “version”);
           int manualBuildColNum = ScraperUtil.getColumnNumber(doc,
autoDeployComponentsTable, “manual”);
           NodeList tableNodeList = doc.getElementsByTagName(“table”);
           Node autoDeployedComponentTable =
tableNodeList.item(autoDeployComponentsTable);
           NodeList autoComponentNodeList = ((Element)
autoDeployedComponentTable).getElementsByTagName(“tr”);
           for (int I = 1; I < autoComponentNodeList.getLength(); i++) {
                   String autoComponentName = ScraperUtil.getCellText(doc,
autoDeployComponentsTable,
                          componentColNum, i).trim();
                   String parentProjectName = ScraperUtil.getCellText(doc,
autoDeployComponentsTable, parentColNum,
                          i).trim();
                   String svnRoot = ScraperUtil.clean(ScraperUtil.getCellText(doc,
autoDeployComponentsTable,
                          svnColNum, i));
                   String versionProp =
ScraperUtil.clean(ScraperUtil.getCellText(doc, autoDeployComponentsTable,
                          versionPropColNum, i));
                   String manuallyBuild =
ScraperUtil.clean(ScraperUtil.getCellText(doc, autoDeployComponentsTable,
                          manualBuildColNum, i));
                   Component halfBuiltComponent = compMap.get(autoComponentName);
                   if (halfBuiltComponent == null) {
                          continue;
                   }
                   if (parentProjectName != null) {

    halfBuiltComponent.setParentComponent(getParentComponent(compMap,
parentProjectName));
                   }
                   if (svnRoot != null) {
                          halfBuiltComponent.setSubversionRoot(svnRoot);
                   }
                   if (versionProp != null) {

    halfBuiltComponent.setParentVersionProperty(versionProp);
                   }
                   if (manuallyBuild != null) {
                          if (manuallyBuild.equals(“Y”)) {
                                  halfBuiltComponent.setManualBuild(true);
                          }
                   }
           }
    }

    private Component getParentComponent(Map<String, Component> compMap, String
parentName) {
           return compMap.get(parentName);
    }
}
```

- **Tagger: Tagger class**

```
      package com.markit.bonds.releasebuilder.tagger;

import java.io.BufferedWriter;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileWriter;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.Xpath;
import javax.xml.xpath.XpathConstants;
import javax.xml.xpath.XpathExpression;
import javax.xml.xpath.XpathFactory;

import org.apache.log4j.Logger;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.XMLSerializer;
import org.tmatesoft.svn.core.SVNDepth;
import org.tmatesoft.svn.core.SVNErrorCode;
import org.tmatesoft.svn.core.SVNException;
import org.tmatesoft.svn.core.SVNURL;
import org.tmatesoft.svn.core.internal.io.dav.DAVRepositoryFactory;
import org.tmatesoft.svn.core.internal.wc.DefaultSVNOptions;
import org.tmatesoft.svn.core.wc.SVNClientManager;
import org.tmatesoft.svn.core.wc.SVNCommitClient;
import org.tmatesoft.svn.core.wc.SVNCopyClient;
import org.tmatesoft.svn.core.wc.SVNCopySource;
import org.tmatesoft.svn.core.wc.SVNRevision;
import org.tmatesoft.svn.core.wc.SVNUpdateClient;
import org.tmatesoft.svn.core.wc.SVNWCUtil;
import org.w3c.dom.Document;
import org.w3c.dom.Node;

import com.markit.bonds.releasebuilder.api.ComponentTagger;
import com.markit.bonds.releasebuilder.api.TaggingCallback;
import com.markit.bonds.releasebuilder.api.TaggingResult;
import com.markit.bonds.releasebuilder.api.exception.TaggingException;
import com.markit.bonds.releasebuilder.api.model.Component;
import com.markit.bonds.releasebuilder.api.model.Release;

public class Tagger implements ComponentTagger {
    private static final long serialVersionUID = 7535523199973690551L;
    private static final Logger log = Logger.getLogger(Tagger.class);
    private SVNClientManager clientManager;
    String url = “https://svn/svn/markit/markitwarehouse/fixedincome/”;
    String svnroot = “/tmp/src/main/resources/test-checkout/”;
    String name = “evb.release.builder”;
    String password = “testing”;
    // String jira = “EVB-3097”;
    Map<Component, Boolean> checkouts;

    public Tagger() {
            checkouts = new HashMap<Component, Boolean>();
    }

    public □pmFile tag(Component c, String jira) throws TaggingException {
            SVNCopyClient copyClient = clientManager.getCopyClient();
            SVNCopySource[] copySource = new SVNCopySource[1];
            String message = jira + “ tagging for release”;
            SVNURL srcUrl;
            try {
                    if (c.firstVersion()) {
                            srcUrl = SVNURL.parseURIEncoded(url +
c.getSubversionPath() + “/trunk/”);
                    } else {
                            srcUrl = SVNURL.parseURIEncoded(url +
c.getSubversionPath() + “/branches/v”
                                          + c.getBranchVersion() + “/”);
                    }
                    SVNURL dstUrl = SVNURL.parseURIEncoded(url +
c.getSubversionPath() + “/tags/v” + c.getVersion());
                    copySource[0] = new SVNCopySource(SVNRevision.HEAD,
SVNRevision.HEAD, srcUrl);
                    copyClient.doCopy(copySource, dstUrl, false, true, true,
message, null);
            } catch (SVNException e) {
                    if
(e.getErrorMessage().getErrorCode().equals(SVNErrorCode.FS_ALREADY_EXISTS)) {
                            return true;
                    } else {
                            throw new TaggingException(c, “Error while tagging”, e);
                    }
            } catch (Exception e) {
                    throw new TaggingException(c, “Error while tagging”, e);
            }
            return false;
    }
    public void commit(Component c, String jira) {
            SVNCommitClient commitClient = clientManager.getCommitClient();
            File[] source = new File[1];
            source[0] = new File(svnroot + c.getSubversionPath());
            String message = jira + “ updated version numbers”;
            try {
                    commitClient.doCommit(source, false, message, null, null, false,
false, SVNDepth.INFINITY);
            } catch (SVNException e) {
                    e.printStackTrace();
            }
    }

    public File □pmFilen(Component c, □pmFile checkoutTag) throws TaggingException
{
            File destination = new File(svnroot + c.getSubversionPath());
            if (checkouts.containsKeyI && checkouts.getI.equals(new Boolean(true)))
{
                    return destination;
            }
            SVNUpdateClient updateClient = clientManager.getUpdateClient();
            SVNURL svnurl;
            try {
                    if (checkoutTag) {
                            svnurl = SVNURL.parseURIEncoded(url +
c.getSubversionPath() + “/tags/v” + c.getVersion());
                    } else if (c.firstVersion()) {
                            svnurl = SVNURL.parseURIEncoded(url +
c.getSubversionPath() + “/trunk/”);
                    } else {
                            svnurl = SVNURL.parseURIEncoded(url +
c.getSubversionPath() + “/branches/v”
                                            + c.getBranchVersion());
                    }
                    updateClient.doCheckout(svnurl, destination, SVNRevision.HEAD,
SVNRevision.HEAD, true, true);
                    checkouts.put(c, new Boolean(true));
                    return destination;
            } catch (SVNException e) {
                    throw new TaggingException(c, “Error while checking out”, e);
            }
    }

    /**
     * Set the version number of the component in the POM
     */
    public void setVersion(Component c, String versionXpath, String version) {
            File file = new File(svnroot + c.getSubversionPath() + “/pom.xml”);
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db;
            try {
                    db = dbf.newDocumentBuilder();
                    Document doc = db.parse(file);
                    XpathFactory factory = XpathFactory.newInstance();
                    Xpath xpath = factory.newXPath();
                    XpathExpression expr = xpath.compile(versionXpath);
                    Node versionNode = (Node) expr.evaluate(doc,
XpathConstants.NODE);
                    versionNode.setTextContent(version);
                    StringBuilder sb = docToStringBuilder(doc);
                    BufferedWriter writer = new BufferedWriter(new
FileWriter(file));
                    writer.write(sb.toString());
                    writer.flush();
                    writer.close();
            } catch (Exception e) {
                    log.error(“Error trying to set the version numver of the
component in the POM”, e);
            }
    }

    public StringBuilder docToStringBuilder(Document doc) {
            StringBuilder □pmFilenames□ = null;
            try {
                    ByteArrayOutputStream stream = new ByteArrayOutputStream();
                    OutputFormat outputformat = new OutputFormat();
                    outputformat.setIndent(4);
                    outputformat.setIndenting(true);
                    outputformat.setPreserveSpace(false);
                    XMLSerializer serializer = new XMLSerializer();
                    serializer.setOutputFormat(outputformat);
                    serializer.setOutputByteStream(stream);
                    serializer.asDOMSerializer();
                    serializer.serialize(doc.getDocumentElement());
                    □pmFilenames□ = new StringBuilder(stream.toString());
            } catch (Exception except) {
                    except.getMessage();
            }
            return □pmFilenames□;
    }

    public void connect() {
            DAVRepositoryFactory.setup();
            DefaultSVNOptions options = SVNWCUtil.createDefaultOptions(true);
            clientManager = SVNClientManager.newInstance(options, name, password);
    }

    public void tagRelease(Release release, TaggingCallback callback) {
            String jira = release.getJiraTicketId();
            clearCheckOutDirectory();
            connect();
            List<Component> components = release.getComponents();
            Map<Component, TaggingException> errorComponents = new
HashMap<Component, TaggingException>();
            // Check out all projects
            for (Component c : components) {
                    try {
                            □pmFilen(c, false);
                    } catch (TaggingException e) {
                            callback.failure(e);
                            errorComponents.put(e.getComponent(), e);
                    }
            }
            // Set versions
            for (Component c : components) {
                    if (errorComponents.containsKeyI) {
                            continue;
                    }
                    if (c.getName().equals(“bonds-parent”)) {
                            System.out.println(“skipping bonds-parent re-
versioning”);
                            continue;
                    }
                    Component target = c.getParentComponent() == null ? c :
c.getParentComponent();
                    String versionXpath = c.getParentComponent() == null ?
“/project/version” : “//”
                                    + c.getParentVersionProperty();
                    setVersion(target, versionXpath, c.getVersion());
            }
            // Commit
            for (Component c : components) {
                    if (errorComponents.containsKeyI) {
                            continue;
                    }
                    commit(c, jira);
            }
            // Tag
            for (Component c : components) {
                    if (errorComponents.containsKeyI) {
                            callback.failure(errorComponents.getI);
                            continue;
                    }
                    try {
                            □pmFile alreadyTagged = tag(c, jira);
                            File checkoutPath = □pmFilen(c, true);
                            if (checkoutPath == null) {
                                    throw new TaggingException(c, “No checkout path
returned after checkout”);
                            }
                            c.setLocalPath(checkoutPath.getAbsolutePath());
                            TaggingResult result = new TaggingResult(release, c,
alreadyTagged);
                            callback.success(result);
                    } catch (TaggingException e) {
                            callback.failure(e);
                    }
            }
            // Check out tags
            callback.taggingComplete();
    }

    public void clearCheckOutDirectory() {
            File dir = new File(svnroot);
            deleteDirectory(dir);
    }

    public □pmFile deleteDirectory(File path) {
            if (path.exists()) {
                    File[] files = path.listFiles();
                    if (files != null) {
                            for (File file : files) {
                                    if (file.isDirectory()) {
                                            deleteDirectory(file);
                                    } else {
                                            file.delete();
                                    }
                            }
                    }
            }
            return (path.delete());
    }
}
```

- **Builder: ComponentBuilderImpl class**

```
package com.markit.bonds.releasebuilder.componentbuilder;

import java.io.File;
import java.io.FilenameFilter;
import java.util.ArrayList;
import java.util.List;

import org.apache.log4j.Logger;

import com.markit.bonds.releasebuilder.api.BuildCallback;
import com.markit.bonds.releasebuilder.api.BuildResult;
import com.markit.bonds.releasebuilder.api.ComponentBuilder;
import com.markit.bonds.releasebuilder.api.exception.BuildException;
import com.markit.bonds.releasebuilder.api.model.Component;
import
com.markit.bonds.releasebuilder.componentbuilder.commandrunner.AbstractCommandRunn
er;
import
com.markit.bonds.releasebuilder.componentbuilder.commandrunner.AbstractCommandRunn
er.CommandException;

/**
 * Builds Mavenised components into RPM files ready for deployment
 *
 * @author Chris.Beach
 */
public class ComponentBuilderImpl implements ComponentBuilder {
    private static final List<String> EXPECTED_INSTALLED_TOOLS = new
ArrayList<String>();
    static {
            EXPECTED_INSTALLED_TOOLS.add(“mvn”);
            EXPECTED_INSTALLED_TOOLS.add(“rpmbuild”);
            EXPECTED_INSTALLED_TOOLS.add(“dos2unix”);
    }
    private static final String COMMAND_MVN = “mvn –Dmaven.test.skip=true clean
test “;
    private static final String COMMAND_RPM = “rpmbuild –bb –target=noarch-linux”;
    private static final String COMMAND_DOS2UNIX = “dos2unix”;
    private static final String FILE_RPM_SPEC = “rpm.spec”;
    private static final String PATH_TARGET = “target”;
    private static final String PATH_RPM = “noarch”;
    private static final Logger log = Logger.getLogger(ComponentBuilderImpl.class);
    private static final AbstractCommandRunner commandRunner =
AbstractCommandRunner.getInstance();

    /**
     * Check that all the required tools are installed
     *
     * @throws Exception
     *             if any tools aren’t found
     */
    public static void checkToolInstallation() throws Exception {
            for (String tool : EXPECTED_INSTALLED_TOOLS) {
                    try {
                            commandRunner.runCommand(“which “ + tool, new
File(“\\”));
                    } catch (CommandException e) {
                            throw new Exception(tool + “ doesn’t appear to be
installed”, e);
                    }
            }
    }

    /**
     * Build the given component using the mvn command, running target/buildrpm.sh
to build RPMs
     *
     * @see
com.markit.bonds.releasebuilder.api.ComponentBuilder#buildComponent(com.markit.bon
ds.releasebuilder.api.model.Component,
     *      java.io.File)
     *
     * @param projectRoot
     *            the directory containing the project and, specifically, its
pom.xml file
     */
    public void buildComponent(String buildWay, Component component, BuildCallback
callback)
                    throws BuildException {
            try {
                    if (component.getLocalPath() == null) {
                            throw new BuildException(component, “No Local path
available for component “ + component);
                    }
                    log.info(“Performing Maven build of “ + component);
                    File projectRoot = new File(component.getLocalPath());
                    commandRunner.runCommand(COMMAND_MVN + buildWay, projectRoot);
                    final List<String> rpmFilenames = buildRpms(component,
projectRoot);
                    callback.success(new BuildResult(component, rpmFilenames));
            } catch (CommandException e) {
                    callback.failure(new BuildException(component, e.getMessage(),
e));
            } catch (BuildException e) {
                    callback.failure(e);
            }
    }

    /**
     * Runs the RPM build command to build a packaged RPM file
     *
     * @return null if no RPM build script is found (assume this isn’t RPM-
deployable)
     */
    private List<String> buildRpms(Component component, final File projectRoot)
throws BuildException {
            File target = new File(projectRoot, PATH_TARGET);
            if (!target.exists()) {
                    log.info(“Building RPM(s)”);
                    throw new BuildException(component, “’target’ directory not
found”);
            }
            File rpmSpecFile = new File(target, FILE_RPM_SPEC);
            if (!rpmSpecFile.exists()) {
                    log.info(“No “ + FILE_RPM_SPEC + “ file, so assuming this isn’t
an RPM-deployable project”);
                    return null;
            }
            try {
                    commandRunner.runCommand(COMMAND_DOS2UNIX + “ “ + FILE_RPM_SPEC,
target);
                    String output = commandRunner.runCommand(
                            COMMAND_RPM + “ –define=\”myversion “ +
component.getVersion() + “\” “ + FILE_RPM_SPEC,
                            target);
                    File rpmPath = new File(target, PATH_RPM);
                    if (!rpmPath.exists()) {
                            throw new BuildException(component, “RPM build didn’t
create “ + PATH_RPM + “ directory”,
                                           output);
                    }
                    // Return a list of RPM files in the target directory
                    File[] rpmFileArray = rpmPath.listFiles(new
RpmFilenameFilter());
                    if (rpmFileArray == null || rpmFileArray.length == 0) {
                            throw new BuildException(component, “No RPM files found
in “ + PATH_RPM, output);
                    }
                    List<String> rpmFilenames = new ArrayList<String>();
                    for (File rpmFile : rpmFileArray) {
                            rpmFilenames.add(rpmFile.getAbsolutePath());
                    }
                    log.info(“Built “ + rpmFileArray.length + “ RPM(s)”);
                    return rpmFilenames;
            } catch (CommandException e) {
                    throw new BuildException(component, e.getMessage(), e);
            }
    }

    /**
     * Filters to RPM files only
     */
    private class RpmFilenameFilter implements FilenameFilter {
            public □pmFile accept(File dir, String name) {
                    return name.toUpperCase().endsWith(“.RPM”);
            }
    }
}
```

- **Deployer: ComponentDeployerImpl class**

```
      package com.markit.bonds.releasebuilder.componentdeployer;

import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.commons.vfs.FileObject;
import org.apache.log4j.Logger;

import com.markit.bonds.releasebuilder.api.ComponentDeployer;
import com.markit.bonds.releasebuilder.api.DeploymentResult;
import com.markit.bonds.releasebuilder.api.exception.DeploymentException;
import com.markit.bonds.releasebuilder.api.model.Component;
import com.markit.bonds.releasebuilder.api.model.Release;
import
com.markit.bonds.releasebuilder.componentdeployer.RemoteCommandRunner.CommandResul
t;
import com.markit.bonds.releasebuilder.componentdeployer.util.RemoteFileUtil;

/**
 * Responsible for deploying built components to hosts and checking which versions
are deployed.
 *
 * @author chris.beach
 */
public class ComponentDeployerImpl implements ComponentDeployer {
    private static final long serialVersionUID = -1944650519281645022L;
    private static final Logger log =
Logger.getLogger(ComponentDeployerImpl.class);
    private static final String PREFIX_PACKAGE = “evb-“;
    private static final String COMMAND_GET_DEPLOYED_VERSIONS = “rpm –qa | grep \””
+ PREFIX_PACKAGE + “\””;
    private static final String COMMAND_START_SERVICE = “/sbin/service [SERVICE]
start”;
    private static final String COMMAND_STOP_SERVICE = “/sbin/service [SERVICE]
stop”;
    private static final String COMMAND_TEST_INSTALL_RPM = “rpm –Uvh –test
[RPM_FILE]”;
    private static final String COMMAND_INSTALL_RPM = “rpm –Uvh [RPM_FILE]”;
    private static final Pattern PATTERN_RPM_PACKAGE_VERSION = Pattern
                    .compile(“([\\w\\-]*)\\-((\\d\\.)*\\d)\\-\\d”);
    private final String unixUsername;
    private final String unixPassword;
    private final String unixHomeDirectory;

    /**
     * Same credentials will be assumed for all hosts
     */
    public ComponentDeployerImpl(String unixUsername, String unixPassword, String
unixHomeDirectory) {
            this.unixUsername = unixUsername;
            this.unixPassword = unixPassword;
            this.unixHomeDirectory = unixHomeDirectory;
    }

    /**
     * Operating on code checked out and built in the project root, send the RPM
file from
     * target/noarch to the given environment host and use SSH to:
     * <ol>
     * <li>check RPM can be installed</li>
     * <li>stop service</li>
     * <li>install RPM</li>
     * <li>start service</li>
     * </ol>
     */
    public DeploymentResult deployComponent(Component component, String
□pmFilenames, String hostname)
                    throws DeploymentException {
            try {
                    File rpmFile = new File(□pmFilenames);
                    FileObject remoteRpm = RemoteFileUtil.copyFileToUrl(rpmFile,
getRemoteUrl(hostname));
                    checkRpmCanBeInstalled(component, remoteRpm, hostname);
                    □pmFile isService = component.getServiceName() != null;
                    if (isService) {
                            stopService(component, hostname);
                    }
                    installRpm(component, remoteRpm, hostname);
                    if (isService) {
                            startService(component, hostname);
                    }
            } catch (Exception e) {
                    throw new DeploymentException(component, e.getMessage(), e);
            }
            return new DeploymentResult(component, false);
    }

    /**
     * @throws Exception
     *              if RPM cannot be installed
     */
    private void checkRpmCanBeInstalled(Component component, FileObject remoteRpm,
String hostname)
                    throws Exception {
            if (remoteRpm == null) {
                    throw new IllegalArgumentException(“Expected remoteRpm”);
            }
            String remoteFilename = remoteRpm.getName().getPath();
            log.info(“Checking if “ + remoteFilename + “ can be installed on “ +
hostname);
            String command = COMMAND_TEST_INSTALL_RPM.replace(“[RPM_FILE]”,
remoteRpm.getName().getPath());
```

```
            CommandResult result =
RemoteCommandRunner.executeRemoteCommand(hostname, unixUsername, unixPassword,
                    command);
            if (!result.succeeded) {
                    throw new DeploymentException(component, “RPM couldn’t be
deployed: “ + result.output);
            }
    }

    private void stopService(Component component, String hostname) throws
IOException {
            String command = COMMAND_STOP_SERVICE.replace(“[SERVICE]”,
component.getServiceName());
            CommandResult result =
RemoteCommandRunner.executeRemoteCommand(hostname, unixUsername, unixPassword,
                    command);
            if (!result.succeeded) {
                    log.warn(“Couldn’t stop service “ + component.getServiceName());
            }
    }

    private void startService(Component component, String hostname) throws
IOException, DeploymentException {
            String command = COMMAND_START_SERVICE.replace(“[SERVICE]”,
component.getServiceName());
            CommandResult result =
RemoteCommandRunner.executeRemoteCommand(hostname, unixUsername, unixPassword,
                    command);
            if (!result.succeeded) {
                    throw new DeploymentException(component, “Could not start
service: “ + result.output);
            }
    }

    private void installRpm(Component component, FileObject remoteRpm, String
hostname) throws IOException,
                    DeploymentException {
            String command = COMMAND_INSTALL_RPM.replace(“[RPM_FILE]”,
remoteRpm.getName().getBaseName());
            CommandResult result =
RemoteCommandRunner.executeRemoteCommand(hostname, unixUsername, unixPassword,
                    command);
            if (!result.succeeded) {
                    throw new DeploymentException(component, “Could not start
service: “ + result.output);
            }
    }

    public Map<Component, Map<String, String>> getDeployedComponentVersions(Release
release) {
            Map<Component, Map<String, String>> deployedComponentVersions = new
HashMap<Component, Map<String, String>>();
            Set<String> hostnames = getUniqueHostnames(release);
            Map<String, Component> components = getComponents(release);
            for (Entry<String, Component> entry : components.entrySet()) {
                    Map<String, String> deployedVersionsOnHost = new HashMap<String,
String>();
                    String componentName = entry.getKey();
                    Component component = entry.getValue();
                    for (String hostname : hostnames) {
                            try {
                                    Map<String, String> packageVersions =
getDeployedEvbPackageVersionsOnHost(hostname);
                                    String componentVersion =
packageVersions.get(componentName);
                                    if (componentVersion != null) {
                                            deployedVersionsOnHost.put(hostname,
componentVersion);
                                    }
                            } catch (Exception e) {
                                    log.error(“Error whilst getting EVB package
versions on host”, e);
                            }
                    }
                    deployedComponentVersions.put(component,
deployedVersionsOnHost);
            }
            return deployedComponentVersions;
    }

    private static Map<String, Component> getComponents(Release release) {
            Map<String, Component> getComponents = new HashMap<String,
Component>();
            for (Component c : release.getComponents()) {
                    getComponents.put(c.getName(), c);
            }
            return getComponents;
    }

    private Set<String> getUniqueHostnames(Release release) {
            Set<String> hostnames = new HashSet<String>();
            if (release.getComponents() == null) {
                    log.debug(“No components in release when trying to get set of
unique hostnames”);
                    return hostnames;
            }
            for (Component component : release.getComponents()) {
                    if (component.getHabitat() == null ||
component.getHabitat().getHostnames() == null) {
                            continue;
                    }
                    for (String hostname : component.getHabitat().getHostnames()) {
                            hostnames.add(hostname);
                    }
            }
            return hostnames;
    }

    /**
     * Get a map<component name, version> from the given host for all EVB
components
     */
    protected Map<String, String> getDeployedEvbPackageVersionsOnHost(String
hostname) throws Exception {
            CommandResult result =
RemoteCommandRunner.executeRemoteCommand(hostname, unixUsername, unixPassword,
                    COMMAND_GET_DEPLOYED_VERSIONS);
            if (!result.succeeded) {
                    throw new Exception(“Couldn’t get deployed RPM version numbers:
“ + result.output);
            }
            String rpmCommandOutput = result.output;
            return getPackageVersions(rpmCommandOutput);
    }

    protected static Map<String, String> getPackageVersions(String rpmText) {
            Map<String, String> map = new HashMap<String, String>();
            String newText = rpmText.replace(PREFIX_PACKAGE, “”);
            Matcher matcher = PATTERN_RPM_PACKAGE_VERSION.matcher(newText);
            while (matcher.find()) {
                    if (matcher.groupCount() < 2) {
                            continue;
                    }
                    String componentName = matcher.group(1);
                    String version = matcher.group(2);
                    map.put(componentName, version);
            }
            return map;
    }

    private String getRemoteUrl(String hostname) {
            return “sftp://” + unixUsername + “:” + unixPassword + “@” + hostname +
unixHomeDirectory;
    }
}
```

- **Web interface: Release_automation class**

```
package com.markit.bonds.releasebuilder.client;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Map;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ChangeHandler;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.logical.shared.ValueChangeEvent;
import com.google.gwt.event.logical.shared.ValueChangeHandler;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.ListBox;
import com.google.gwt.user.client.ui.RadioButton;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextArea;
import com.google.gwt.user.client.ui.TextBox;
import com.markit.bonds.releasebuilder.api.model.Component;
import com.markit.bonds.releasebuilder.api.model.Release;
import com.markit.bonds.releasebuilder.client.BuildTask.BuildTaskCallback;
import
com.markit.bonds.releasebuilder.client.DeploymentTask.DeploymentTaskCallback;
import com.markit.bonds.releasebuilder.client.TaggingTask.TaggingTaskCallback;

public class Release_automation implements EntryPoint, TaggingTaskCallback,
BuildTaskCallback,
            DeploymentTaskCallback {
    /**
     * Create a remote service proxy to talk to the server-side Greeting service.
     */
    private final AutomationServiceAsync automationSvc =
GWT.create(AutomationService.class);
    private Release releaseFromWiki;
    private Release releaseWithCheckedComponents;
    private Map<Component, Map<String, String>> deployedVersions;
    final ListBox releaseNameMenu = new ListBox(false);
    final static ListBox releaseEnvMenu = new ListBox(false);
    final ComponentTable componentTable = new ComponentTable();
    final static String DEFAULTENV = “QA”;
    List<Component> releaseComponents = new ArrayList<Component>();
    List<Component> taggedComponentsResult = new ArrayList<Component>();
    Button refreshButton = new Button(“refresh”);
    Button startButton = new Button(“Start”);
    Button deployButton = new Button(“Deploy”);
    TextArea rpmList = new TextArea();
    TextBox jiraTicket = new TextBox();
    RadioButton install = new RadioButton(“radioButton”, “Install”);
    RadioButton deploy = new RadioButton(“radioButton”, “Deploy”);
    private String buildWay;
    static {
            releaseEnvMenu.insertItem(“QA”, 0);
            releaseEnvMenu.insertItem(“PROD”, 1);
            releaseEnvMenu.insertItem(“DEV”, 2);
            releaseEnvMenu.insertItem(“DR”, 3);
            releaseEnvMenu.setVisibleItemCount(1);
    }

    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {
            GWT.log(“Module loaded”);
            // set up release name menu
            releaseNameMenu.setVisibleItemCount(1);
            releaseNameMenu.setWidth(“300px”);
            RootPanel.get(“releaseNameMenu”).clear();
            RootPanel.get(“releaseNameMenu”).add(releaseNameMenu);
            // set up release environment menu
            RootPanel.get(“releaseEnvironmentMenu”).add(releaseEnvMenu);
            releaseEnvMenu.addChangeHandler(new ChangeHandler() {
                    public void onChange(com.google.gwt.event.dom.client.ChangeEvent
event) {
                            for (int I = 0; I < releaseEnvMenu.getItemCount(); i++)
                                    if (releaseEnvMenu.isItemSelected(i)) {

    loadRelease(releaseEnvMenu.getItemText(i),
    releaseNameMenu.getItemText(releaseNameMenu.getSelectedIndex()));
                                    }
                     }
            });
            RootPanel.get("releaseNameMenuText").clear();
            RootPanel.get("releaseNameMenuText").add(new Label("Choose release
name:"));
            // set up deploy button
            // deployButton.setVisible(false);
            // deployButton.addStyleName("deploy");
            // RootPanel.get("deployButton").add(deployButton);
            refreshButton.addClickHandler(new ClickHandler() {
                    public void onClick(ClickEvent event) {
                            new DeploymentTask(releaseWithCheckedComponents,
automationSvc, componentTable,
                                         Release_automation.this);
                    }
            });
            // set up component table
            RootPanel.get("componentTable").add(componentTable);
            componentTable.setDefaultHeadings();
            componentTable.clear();
            // set up radio button
            install.setVisible(false);
            deploy.setVisible(false);
            RootPanel.get("radioButton").add(install);
            RootPanel.get("radioButton").add(deploy);
            install.addValueChangeHandler(new ValueChangeHandler<Boolean>() {
                    public void onValueChange(ValueChangeEvent<Boolean> event) {
                            if (event.getValue()) {
                                    buildWay = "install";
                                    startButton.setVisible(true);
                            }
                    }
            });
            deploy.addValueChangeHandler(new ValueChangeHandler<Boolean>() {
                    public void onValueChange(ValueChangeEvent<Boolean> event) {
                            if (event.getValue()) {
                                    buildWay = "deploy";
                                    startButton.setVisible(true);
                            }
                    }
            });
            // set up jira ticket □pmFile
            jiraTicket.setVisible(false);
            RootPanel.get("jiraTicket").add(jiraTicket);
            jiraTicket.setWidth("200px");
            jiraTicket.addChangeHandler(new ChangeHandler() {
                    public void onChange(com.google.gwt.event.dom.client.ChangeEvent
event) {
                            deploy.setVisible(true);
                            install.setVisible(true);
                    }
            });
            // set up refresh button
            refreshButton.setVisible(false);
            refreshButton.addStyleName("refresh");
            RootPanel.get("refreshButton").add(refreshButton);
            refreshButton.addClickHandler(new ClickHandler() {
                    public void onClick(ClickEvent event) {
                            RootPanel.get("rpmListText").clear();
                            releaseEnvMenu.setVisible(true);
                            loadRelease(DEFAULTENV,
releaseNameMenu.getItemText(releaseNameMenu.getSelectedIndex()));
                    }
            });
            // set up start button
            startButton.setVisible(false);
            startButton.addStyleName("startButton");
            RootPanel.get("startButton").clear();
            RootPanel.get("startButton").add(startButton);
            startButton.addClickHandler(new ClickHandler() {
                    public void onClick(ClickEvent event) {
                            releaseWithCheckedComponents =
□pmFilenames□hcheckedComponents(releaseFromWiki);
                            RootPanel.get("allButton").clear();
```

```
                            RootPanel.get(“clearButton”).clear();
                            startButton.setVisible(false);
                            if (releaseWithCheckedComponents != null) {
                                    new TaggingTask(releaseWithCheckedComponents,
automationSvc, componentTable,
                                                    Release_automation.this);
                                    deploy.setVisible(false);
                                    install.setVisible(false);
                            }
                    }
            });
            RootPanel.get(“rpmList”).add(rpmList);
            rpmList.addStyleName(“rpmList”);
            rpmList.setVisible(false);
            getReleaseNames();
    }

    /**
     * Load a list of all releases, then populate the UI with the details, then
load the current
     * release
     */
    private void getReleaseNames() {
            automationSvc.getReleaseNames(new AsyncCallback<Collection<String>>() {
                    public void onFailure(Throwable caught) {
                            caught.printStackTrace();
                    }

                    public void onSuccess(final Collection<String> result) {
                            getCurrentRelease(result);
                            releaseNameMenu.addChangeHandler(new ChangeHandler() {
                                    public void
onChange(com.google.gwt.event.dom.client.ChangeEvent event) {
                                            RootPanel.get(“rpmListText”).clear();
                                            releaseNameMenu.setVisibleItemCount(1);
                                            loadRelease(DEFAULTENV,

    releaseNameMenu.getItemText(releaseNameMenu.getSelectedIndex()));
                                    }
                            });
                    }
            });
    }

    /**
     * Get all the release name from server and add them into menu bar, set default
to current
     * release Then add change handler to the corresponding release name
     */
    private void getCurrentRelease(final Collection<String> result) {
            automationSvc.getCurrentReleaseName(new AsyncCallback<String>() {
                    public void onFailure(Throwable caught) {
                    }

                    public void onSuccess(String currentRelease) {
                            loadRelease(DEFAULTENV, currentRelease);
                            releaseNameMenu.insertItem(currentRelease, 0);
                            for (final String releaseName : result) {
                                    int I = 1;
                                    if (releaseName.contains(“current”)) {
                                            continue;
                                    }
                                    releaseNameMenu.insertItem(releaseName, i);
                                    I += 1;
                            }
                    }
            });
    }

    // Display table for corresponding release name in menu bar
    private void loadRelease(final String releaseEnv, final String releaseName) {
            rpmList.setText(null);
            rpmList.setVisible(false);
            automationSvc.getRelease(releaseEnv, releaseName, new
AsyncCallback<Release>() {
                    public void onFailure(Throwable caught) {
                    }
```

```
                   public void onSuccess(Release result) {
                          refreshButton.setVisible(false);
                          releaseFromWiki = result;
                          populateComponentTable();
                          if (releaseFromWiki.getComponents().size() > 0)
                                 jiraTicket.setVisible(true);
                          if (jiraTicket.getText().length() > 0) {
                                 startButton.setVisible(true);
                          }
                          RootPanel.get("jiraTicketText").clear();
                          RootPanel.get("jiraTicketText").add(new Label("Enter
jira ticket:"));
                   }
           });
    }

    private void populateComponentTable() {
           componentTable.clear();
           componentTable.setDefaultHeadings();
           componentTable.getRowFormatter().setVisible(0, false);
           new DeploymentTask(releaseFromWiki, automationSvc, componentTable,
Release_automation.this);
           componentTable.populateWithRelease(releaseFromWiki, deployedVersions);
    }

    private Release □pmFilenames□hcheckedComponents(Release fullRelease) {
           return new Release(fullRelease.getName(),
componentTable.getCheckedComponentList(fullRelease),
                          jiraTicket.getText());
    }

    public void taggingComplete(List<Component> taggedComponents) {
           if (!taggedComponents.isEmpty()) {
                   taggedComponentsResult = taggedComponents;
                   new BuildTask(buildWay, taggedComponentsResult, automationSvc,
componentTable,
                                 Release_automation.this);
           } else if (taggedComponents.isEmpty()) {
                   refreshButton.setVisible(true);
           }
    }

    public void buildComplete(List<Component> builtComponents, List<String>
rpmFilenames) {
           if (rpmFilenames.size() > 0) {
                   rpmList.setVisible(true);
                   StringBuilder sb = new StringBuilder();
                   for (String filename : rpmFilenames) {
                          sb.append(filename);
                          sb.append("\n");
                   }
                   rpmList.setText(sb.toString());
                   RootPanel.get("rpmListText").clear();
                   RootPanel.get("rpmListText").add(new Label("RPM files"));
           }
           if (rpmFilenames != null) {
                   // deployButton.setVisible(true);
           }
    }

    public void deploymentComplete() {
           refreshButton.setVisible(true);
    }

    public void getCurrentVersions(Map<Component, Map<String, String>> results) {
           deployedVersions = results;
    }
}
```

- **Web interface: AutomationServiceImpl class**

```
package com.markit.bonds.releasebuilder.server;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

import org.apache.log4j.Logger;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.markit.bonds.coreutil.types.StringUtil;
import com.markit.bonds.releasebuilder.api.BuildCallback;
import com.markit.bonds.releasebuilder.api.BuildResult;
import com.markit.bonds.releasebuilder.api.ComponentBuilder;
import com.markit.bonds.releasebuilder.api.ComponentDeployer;
import com.markit.bonds.releasebuilder.api.ComponentTagger;
import com.markit.bonds.releasebuilder.api.ConfluenceScraper;
import com.markit.bonds.releasebuilder.api.DeploymentResult;
import com.markit.bonds.releasebuilder.api.TaggingCallback;
import com.markit.bonds.releasebuilder.api.TaggingResult;
import com.markit.bonds.releasebuilder.api.exception.BuildException;
import com.markit.bonds.releasebuilder.api.exception.DeploymentException;
import com.markit.bonds.releasebuilder.api.exception.TaggingException;
import com.markit.bonds.releasebuilder.api.model.Component;
import com.markit.bonds.releasebuilder.api.model.Release;
import com.markit.bonds.releasebuilder.client.AutomationService;
import com.markit.bonds.releasebuilder.componentbuilder.ComponentBuilderImpl;
import com.markit.bonds.releasebuilder.componentdeployer.ComponentDeployerImpl;
import com.markit.bonds.releasebuilder.confluencescraper.ConfluenceScraperImpl;
import com.markit.bonds.releasebuilder.tagger.Tagger;

/**
 * The server side implementation of the RPC service.
 */
@SuppressWarnings(“serial”)
public class AutomationServiceImpl extends RemoteServiceServlet implements
AutomationService, BuildCallback,
            TaggingCallback {
    private static final Logger log =
Logger.getLogger(AutomationServiceImpl.class);
    List<BuildResult> outstandingBuildResults = new ArrayList<BuildResult>();
    List<TaggingResult> outstandingTaggingResults = new ArrayList<TaggingResult>();
    List<TaggingException> taggingExceptions = new ArrayList<TaggingException>();
    List<BuildException> BuildExceptions = new ArrayList<BuildException>();
    Map<Component, List<String>> deployedVersionLists = new HashMap<Component,
List<String>>();
    private final String unixUsername = “release”;
    private final String unixPassword = “eight8”;
    private final String unixHomeDirectory = “/home/release”;

    public String getCurrentReleaseName() {
            ConfluenceScraper scraper = new ConfluenceScraperImpl();
            return scraper.getCurrentReleaseName();
    }

    public Release getRelease(String releaseEnv, String releaseName) throws
IllegalArgumentException {
            ConfluenceScraper scraper = new ConfluenceScraperImpl();
            return scraper.getRelease(releaseEnv, releaseName);
    }

    public Collection<String> getReleaseNames() {
            ConfluenceScraper scraper = new ConfluenceScraperImpl();
            return scraper.getReleaseNames();
    }

    /**
     * Takes an API Release object and converts to a GWT model object suitable for
use in
     * client-server interactions
     */
    public void tagRelease(Release release) {
            ComponentTagger tagger = new Tagger();
            try {
                    tagger.tagRelease(release, this);
            } catch (TaggingException e) {
                    e.printStackTrace();
            }
    }

    public List<TaggingResult> getOutstandingTaggingResults() {
            List<TaggingResult> taggingResults = new
ArrayList<TaggingResult>(outstandingTaggingResults);
            outstandingTaggingResults.clear();
            return taggingResults;
    }

    public List<TaggingException> getFailureTaggingResults() {
            List<TaggingException> results = new
ArrayList<TaggingException>(taggingExceptions);
            taggingExceptions.clear();
            return results;
    }

    public void success(TaggingResult result) {
            log.info(“GWT server got tagging result for component “ +
result.getComponent());
            outstandingTaggingResults.add(result);
            System.out.println(result.getComponent().getName() + “ tag success”);
    }

    public void failure(TaggingException exception) {
            System.out.println(“tag failure”);
            exception.setTrace(StringUtil.errorToString(exception));
            taggingExceptions.add(exception);
    }

    public void taggingComplete() {
            System.out.println(“tag complete”);
    }

    public void buildComponent(String buildWay, Component component) throws
BuildException {
            ComponentBuilder builder = new ComponentBuilderImpl();
            builder.buildComponent(buildWay, component, this);
    }

    public List<BuildResult> getOutstandingBuildResults() {
            List<BuildResult> buildResults = new
ArrayList<BuildResult>(outstandingBuildResults);
            outstandingBuildResults.clear();
            log.info(“Returning “ + buildResults.size() + “ build results from
server to client”);
            return buildResults;
    }

    public List<BuildException> getFailureBuildResults() {
            List<BuildException> results = new
ArrayList<BuildException>(BuildExceptions);
            BuildExceptions.clear();
            return results;
    }

    public void success(BuildResult result) {
            System.out.println(result.getComponent().getName() + “ build success”);
            log.info(“GWT server got build result for component “ +
result.getComponent());
            outstandingBuildResults.add(result);
    }

    public void failure(BuildException exception) {
            System.out.println(“build failure”);
            exception.setTrace(StringUtil.errorToString(exception));
            BuildExceptions.add(exception);
    }

    public void buildComplete() {
            System.out.println(“build complete”);
    }

    public Map<Component, List<String>> getDeployedComponentVersionLists(Release
release) {
            ComponentDeployerImpl deployer = new
ComponentDeployerImpl(unixUsername, unixPassword,
                          unixHomeDirectory);
            Map<Component, Map<String, String>> deployedVersions =
deployer.getDeployedComponentVersions(release);
            List<String> versions = new ArrayList<String>();
            Set<Entry<Component, Map<String, String>>> compSet =
deployedVersions.entrySet();
            for (Entry<Component, Map<String, String>> entry : compSet) {
                    Component c = entry.getKey();
                    for (Entry<String, String> versionEntry :
entry.getValue().entrySet()) {
                            versions.add(versionEntry.getValue());
                    }
                    deployedVersionLists.put(c, versions);
            }
            return deployedVersionLists;
    }

    public Map<Component, Map<String, String>> getDeployedComponentVersions(Release
release) {
            ComponentDeployerImpl deployer = new
ComponentDeployerImpl(unixUsername, unixPassword,
                            unixHomeDirectory);
            return deployer.getDeployedComponentVersions(release);
    }

    public DeploymentResult deployComponent(Component component, String
□pmFilenames, String hostname) {
            ComponentDeployer deployer = new ComponentDeployerImpl(unixUsername,
unixPassword, unixHomeDirectory);
            try {
                    deployer.deployComponent(component, □pmFilenames, hostname);
            } catch (DeploymentException e) {
                    e.printStackTrace();
            }
            return null;
    }
}
```

Appendix E: Project Division

Since we divided the project into Scraper, Tagger, Builder and Deployer as well as Web interface, two of the EVB team developers were involved in the design and implementation of Release Builder besides me.

Chris Beach is mainly responsible for implementing Builder and Deployer. Dmitri is allocated to do Tagger. My job is to achieve the functionalities of Scraper and Web interface.

We worked together in the initial design phase. We mapped out a problem statement and project timeline at the initial stage. Requirements and design were debated amongst the team and then collated and categorised in the Confluence page during the first two weeks. In implementing design, we specifically focused on our own part but worked very closely with each other to make sure codes written are standardized and easy for others to understand.

Appendix F: Content of CD

The CD contains the components of:

- Project File:
  The project file contains six Java project files:
  release-builder-component-api
  release-builder-confluence-scraper
  release-builder-component-tagger
  release-builder-component-builder
  release-builder-component-deployer
  release-builder-webapp
- Project_report.doc
- Project_report.pdf

Bibliography

[1] Adam Tacy and Robert Hanson. Gwt in Action, Manning Publications, 2007

[2] Robert Cooper and Charles Collins. GWT in Practice, Manning Publications, 2008

[3] Dan Sanderson. Programming Google App Engine, O' Reilly Media, 2009

[4] Ryan Dewsbury. Google Web Toolkit Applications, Prentice Hall PTR, 2007

[5] Michael Bays. Software Release Methodology, Prentice Hall, 1999

[6] Ben Collins-Sussman and Brian Fitzpatrick. Version Control with Subversion, O' Reilly Media, 2008

[7] Sonatype Company. Maven: The Definitive Guide, O' Reilly Media, 2008

[8] Daniel Berlin and Garrett Rooney. Practical Subversion, Apress, 2006

[9] Mike Mason. Pragmatic Version Control: Using Subversion, Pragmatic Bookshelf, 2006

[10] Johannes Link and Peter Frohlich. Unit Testing in Java: How Tests Drive the Code, 2003

[11] Bruce Eckel. Thinking in Java, Prentice Hall PTR, 2006

[12] J.B. Rainsberger and Scott Stirling. JUnit Recipes: Practical Methods for Programmer Testing, 2004