HUBcheck: Check the hub

Derrick S. Kearney
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Derrick Kearney

Entitled
Hubcheck: Check the Hub

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

SAMUEL P. MIDKIFF	
MARY L. COMER	
MILIND KULKARNI	
T. N. VIJAYKUMAR	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): SAMUEL P. MIDKIFF

Approved by: Michael R. Melloch 04/24/2015

Head of the Department Graduate Program Date

HUBCHECK: CHECK THE HUB

A Thesis

Submitted to the Faculty

of

Purdue University

by

Derrick S. Kearney

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2015

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I would like to express my gratitude to everyone who encouraged and worked with me through this process of exploring software based automation.

I would like to thank my family for pushing me to always pursue education.

I would like to thank my committee members, Professor Samuel Midkiff, Professor Mary Comer, Professor Milind Kulkarni, and Professor T. N. Vijaykumar, for teaching me about research and challenging my scientific thought process.

I would like to thank Dr. Mike McLennan, Mr. George Howlett, and the HUBzero Team for providing time, resources, and feedback while working on this project.

I would like to thank the members of the Purdue Perl Mongers and GLOSSY for providing a platform where people can express their software related experiences and idea.

I would like to thank the staff and faculty of the College of Electrical and Computer Engineering, especially the members of the Graduate Office who have helped me start, continue, and finish my graduate education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Kearney, Derrick S. M.S.E.C.E., Purdue University, May 2015. HUBcheck: Check the Hub. Major Professor: Sam Midkiff.

The HUBzero Platform is a framework for building websites, referred to as "hubs," that promote research communities through online simulation, data management, and collaboration. With each software release, the HUBzero Team dedicates weeks of team members' time toward manually testing, fixing, and retesting hub components. The unique mixture of environments that make up a hub makes using existing automated testing solutions hard and shifts the burden of testing to humans, promoting variation, spot checking of fixes, and other shortcuts to avoid the high cost of completely retesting the system. With over twenty hubs being actively managed by the HUBzero Team, manually testing each one after a software update is resource and time prohibitive.

The HUBcheck library, a collection of Python modules backed by Selenium WebDriver and Paramiko, was built to help developers write automation scripts for HUBzero websites and the Debian Linux based virtual containers hosting the hub's simulation tools. Today, the HUBzero Team is using HUBcheck to perform automated regression testing on all of its production hubs, regularly testing areas of the hub that were previously overlooked. In this document, we investigate how HUBcheck works, introduce three new design patterns that make writing page object based automation easier, and show how the use of HUBcheck has helped reduce the number of misconfigured systems during a one year period of hub upgrades.

1. INTRODUCTION

HUBzero is an open source software platform for building websites, or hubs, that support collaborative research, science, and education. Like all software, the platform has bugs being introduced and fixed with every release. Bugs in hubs can be hard to find, and once fixed can be reintroduced in a later release, which make them good candidates for automated testing. The HUBzero platform is a dual environment system consisting of a website that uses pluggable components to provide services to users, and middleware that provides access to Linux based containers where simulation tools are developed and run. Due to its complexity, testing on the hub introduces unique challenges involving the interactions between the web server and middleware that cannot be addressed by currently available testing software. The HUBcheck library was created to address these challenges and provide hub developers with a single toolkit for performing end user testing on hubs managed by the HUBzero Team at Purdue University.

1.1 Origins of Hub Testing

In 2005, the HUBzero Team supported their first hub, nanoHUB.org. In 2007, the number of hubs in production had risen to four, and by 2012, the team was supporting twenty five hubs. The release of the open source version allowed others to begin launching self-supported hubs. Each deployed hub started from a single core version of the software and quickly bloomed into its own system with the standard set of hub components and slightly different configurations and content.

Despite the best intentions of the HUBzero Team, hub software was, and still is, being released with bugs. The same economies of scale that allow the group to rapidly deploy quality software work against the group when combating erroneous software. A single bug that is discovered on one hub website is usually also on numerous other hub websites. In the past, different upgrade schedules and a lack of upgrade documentation made it hard to tell which hubs had one of the many bugs reported to the team. Additionally, the lack of a test suite made it difficult to find bugs before the software was released.

Before HUBcheck, the testing of hub components was performed by hand, making it a time consuming and error prone process. The manual testing promoted variation in tests performed. Due to its repetitive nature, each iteration of testing could be performed a different way, or not performed if forgotten. The escalating commitment needed to setup and perform tests encouraged spot checking of bug fixes instead of a broad retesting of components.

1.2 A Hub Specific Testing Solution

The hub is a controlled environment where the developers understand how the components are supposed to work. HUBcheck, a set of hub automation libraries, allows developers to take advantage of this knowledge and perform targeted end user tests for hub components that the industry standard tools do not support. HUBcheck provides a collection of tools which can be used to help automate the testing of how users interact with hub components that would otherwise be performed by hand. When compared to testing by hand, HUBcheck reduces testing time, increases test coverage, and provides a reliable way to reproduce errors.

The HUBcheck libraries focus on two types of automation, hub access through a secure shell (SSH) and hub access through a web browser. Building on top of Python's Paramiko libraries and using ideas from Tcl's Expect, HUBcheck provides functions to easily automate access to different environments through SSH. This enables developers to test lower level system setup and configuration, for example, within a tool session container. Similarly, HUBcheck uses the Selenium library to automate user interactions with the hub website. HUBcheck libraries provide abstractions of hub components, that can be used to write maintainable automation scripts. When combined, these two automation models can be used to verify that a hub is running as it was intended. By using HUBcheck to write test cases for hub components, a hub can be verified in under a day. The decrease in testing time and increase in tested components encourages the adoption of automated testing earlier in the development cycle, where errors cost less to fix.

This thesis describes the inner workings of HUBcheck and how it is utilized by the HUBzero Team to track and reduce the number of software defects in the HUBzero Platform. The thesis starts by reviewing related work in Chapter 2 and providing background information about how the hub works in Chapter 3. Chapter 4 discusses the types of problems commonly seen on the hub and the reasons they can be hard to test. In Chapter 5, we look into the external libraries that HUBcheck depends on to enable developers to automate processes. Chapter 6 dives into HUBcheck's modules, explaining how it builds upon tools like Selenium WebDriver and Paramiko to provide a library for hub automation and testing. Chapter 7 continues the HUBcheck discussion by introducing best practices for building maintainable test cases. Lastly, empirical data from using HUBcheck on hubs managed by the HUBzero Team is presented in Chapter 8 and future work for the HUBcheck project is laid out in Chapter 9.

2. RELATED WORK

Automated tools exist to aid in the testing of web applications and cover popular testing topics including performance, security, functionality, usability, and user interfaces. One popular class of automated web testing tools are *web crawlers*, such as Crawljax [1] and IBM Security AppScan [2]. Web crawlers traverse a web application, performing tests based on the existence of interactive widgets (text boxes, links, and buttons) on web pages.

Using a web crawler can require very little coordination from a developer. The crawler's semi-directed exploration of the web application allows it to treat the application like a black box when checking for bugs. Web crawlers navigate to reachable web pages, review the available widgets on the web page, attempt actions, and document the results. This makes them ideal for building site maps, testing navigation, checking for bad links, and performing security scans across generic web pages. The use of a web crawler is not without cost. Web crawlers can take a long time to run because their goal is to exhaust every path through a graph that represents a web application under test. The upside is that they have the opportunity to run a large number of tests on the web application, in an automated fashion, with little initial investment of time from the developer.

For web crawlers, knowledge of how the web application works and its limits are usually discovered at runtime, which can require a time consuming, exhaustive search of the problem space. Even after the web application has been "learned," there is no authority to determine if the learned behavior is the correct behavior. For example, an important part of building a successful hub is allowing content developers to create and deploy their own resources. The first step of contributing a resource to the hub is to fill out a web form describing the content that is being contributed. The form generally contains text boxes and drop down lists, some of which are required or have special restrictions on accepted values. A web crawler would be able to interact with the widgets in the web form, test navigation of links on the page, and test the security of inputting values into the text boxes in the form. Without additional knowledge provided by the developer, the web crawler would have a hard time determining if a widget was supposed to be required or optional, or confirming that each widget's input validation was working properly. Exploring every path through the web application graph is good for whole system testing, but when it comes to targeted testing of a specific set of actions, another class of tools is available.

This second class of automated web testing tools are *test recorders*, including Selenium WebDriver [3], Webking, and Sahi [4]. Test recorders have explicit web application navigation based on actions previously recorded by the developer. Recording the actions for these tools to perform is a time consuming, manual process compared to the more automatic web crawlers, but can produce specific, targeted tests. In the case of Selenium WebDriver, these recordings can be programmed into a script and coordinated with actions performed by other remotely controlled applications such as a terminal shell.

Using a test recorder based tool, hub developers could record or program the steps of filling in the registration form of the resource contribution process, described in the previous example. Widget input validation can be exercised by running the automated script with different, developer specified, input values. By allowing the developer to choose the input values, knowledge of the widget's restrictions are embedded within the collection of tests. While web crawler based tools would need to discover a widget's restrictions, test recorder based tools have the restrictions defined by the hub developer.

Web crawler and test recorder based tools are designed for testing web pages. On the hub, content developers and users interact with web pages, but they also interact with a component that is not a web page, called a tool session container. Tool session containers are Debian Linux based OpenVZ containers that allow content developers to build and deploy simulation tools on the hub. Testing within the tool session container requires accessing it through a secure shell (SSH) connection. Traditional web crawler and test recorder based tools are not designed for this, and, on their own, cannot be used for this purpose.

3. HUBZERO

3.1 The HUBzero Platform

The HUBzero Platform [5] is an open source software platform designed to meet the needs of researchers and educators through the use of dynamic web sites and interactive simulation tools. Websites based on the HUBzero software stack, also known as hubs, power science by supporting research, education, and collaboration.

Three groups of users interact with the hub environment. Hub developers build the hub by writing website components, creating the middleware and application toolkits, and monitoring content creation channels. Content developers produce the presentations, articles, software simulators, and other material that attracts people to the hub. While hub users are the consumers of content available on the hub, they are often also content developers.

There are three features that separate a hub from other websites available on the internet:

1. Simulation Tools - running software simulations from a web browser
2. Content Sharing - allowing users to upload and download content from the community
3. Support for Collaboration - helping people work together and learn from each other

3.1.1 Simulation Tools

Hubs allow content developers to build and deploy software applications as tools with graphical user interfaces, available for hub users to run inside of a web browser.

The HUBzero platform provides the content developers with a virtual Linux environment, called a tool session container, where they can develop simulation tools that support their science. Inside the tool session container, developers are able to access high performance computing resources and incorporate output from visualization servers in their tool. Once created, these simulation tools can be published as resources on the hub, where hub users can launch and interact with them through a web browser. Publishing simulation tools on the hub removes the burden on users of downloading and installing the software on their own computer, fighting with compiler errors, and acquiring access to restricted resources. Simulation tools available on the hub provide a seamless, end-to-end experience for users.

3.1.2 Sharing Content

Hubs allow content developers to upload datasets, presentations, teaching materials, publications, simulation tools (as mentioned in section 3.1.1), and other types of materials. Once published on the hub, these materials can be shared with the world. Hubs are community driven and the ability for users to influence and shape the community is a key feature of the hub.

3.1.3 Support for Collaboration

Hubs allow users to work together and form communities. Users can create groups within the hub to cultivate special interests, or create projects with document repositories used to organize and track changes in data. Hubs support other methods to collaborate with the community through the "Questions and Answers" forums, where users can ask and respond to questions posted by other members, and through wishlists, where users can suggest changes to help improve the hub.

A hub revolves around a scientific community, just as science itself does. The HUBzero Team hosts over 20 hubs, supporting scientists researching nanotechnology, earthquake engineering, healthcare systems, pharmaceutical manufacturing, cancer,

Table 3.1: List of 2013's Largest hubs sorted by number of users

Hub	# Users	# Visitors
nanohub.org [6]	269,461	557,663
nees.org [7]	78,177	265,075
pharmahub.org [8]	24,213	35,198
vhub.org [9]	13,841	38,600
stemedhub.org [10]	5,104	15,943
ccehub.org [11]	4,346	18,431
habricentral.org [12]	3,760	48,040
molecularhub.org [13]	2,675	15,995
purr.purdue.edu [14]	2,636	15,281
iemhub.org [15]	2,421	12,821
c3bio.org [16]	2,360	15,537
cleerhub.org [17]	1,257	7,988
drinet.hubzero.org [18]	1,082	10,123
iashub.org [19]	1,062	13,316
diagrid.org [20]	789	8,238
memshub.org [21]	781	5,851
geoshareproject.org [22]	600	6,630
catalyzecare.org [23]	460	8,050

volcanoes, biomass energy, and more. Together, these hubs comprise over 400,000 users and over a million visitors per year. Table 3.1 provides a breakdown of users and visitors for the largest hubs supported by the HUBzero Team. User counts include registered accounts, unregistered users with a unique IP address or hostname that remained active while visiting the site for at least 15 minutes, and uniquely identifiable unregistered users who download a resource from the website. Visitors are identified by a unique IP address or hostname.

3.2 Hub Components

The HUBzero platform uses a plugin based architecture. Customization and features are added through plugin extensions named *Components*. Out of the box, a hub comes with components that are designed to support research, education and

Table 3.2: Hub components are built to power science

Component	Research	Education	Collaboration
Simulation Tools	X	X	X
Courses		X	
Resources	X	X	
Groups	X		X
Projects	X		X
Databases	X		X
Questions and Answers		X	

collaboration. Table 3.2 shows a few of the more popular components and how they contribute to the hub powering Science.

3.2.1 Simulation Tools

The HUBzero platform supports the publishing of software simulation tools with graphical user interfaces. Hubs follow a tool contribution process which outlines how users can develop, install and publish their own software. This process allows scientists and researchers to disseminate their software on the hub. Simulation tools run in an OpenVZ [24] container called a tool session container. Tool session containers are hosted on the hub, and have an X11 server that is projected to the user's desktop through a VNC [25] connection. On hubs hosted by the HUBzero Team, tool session containers have access to visualization servers and national grid computing resources.

Tool session containers on hosted hubs have access to visualizations servers that can handle VTK data, PYMOL data, and the home grown nanoVIS data format. Users working in these tool session containers can also submit jobs to various grid computing resources including Diagrid [26], Open Science Grid [27] and XSEDE [28]. Access to these premium computing resources allows researchers to build simulation tools for users who may not have access to the powerful machines needed to run parallel cluster jobs or visually represent large datasets that are produced as results.

3.2.2 Resources

The Resources component provides hub content developers with a way to upload their own online presentations, publications, animations, and other downloadable content. Contributing a resource is similar to contributing a simulation tool. The content developer provides a title, description, citation, and author information for the resource being contributed. After being reviewed, the resource is published on the hub. These materials, generated by the hub's community, are an important aspect of the hub. When content developers upload resources, it promotes the education of users and helps spread the work and ideas of community members.

3.2.3 Courses

The Courses component allows educators to upload various course material including lectures, tests, quizzes, homework and notes, and organize them in a timeline format for dissemination as a course. Courses can also pull in published hub resources and simulations tools. Each course can support multiple offerings, or versions of the course, and each offering can support multiple sections running on different schedules. The Courses component has an interface for hub users where they can track their progress, view lectures, take quizzes, and download homework. The Courses component directly supports the educational goals of the hub.

3.2.4 Groups

Hub users can form specialized communities on the hub by creating a group using the Groups component. Membership to user created groups can be opened to the public, restricted to certain people, or completely private. Within a group, members can upload resources, share content, start conversations and host projects. Groups promote collaboration and help teams of people share research.

3.2.5 Projects

The Projects component on the hub makes collaborating with other hub users easy. Similar to the Groups component, Projects lets users manage a team of users and collaborate. Projects also provide users with management tools like to-do lists, notes, a Git based file repository, and connections to cloud resources such as Google Drive and DropBox. Projects were created to help ease the process of collaborating on funding proposals, writing research papers, and managing data.

3.2.6 Databases

The Databases component allows hub users to quickly populate and search a database based on data from a spreadsheet or file. Using the Databases component, users can create views, plot, and combine data with maps. Data from databases can also be shared with simulation tools for further processing.

3.2.7 Questions and Answers

The Questions and Answers component promotes education and collaboration between hub users. Using this component, users can pose questions to the community and get responses. The best questions are voted up and the best answers can receive a reward of points that can be used on the hub website.

4. THE HUBCHECK PROBLEM SPACE

Problems on the hub creep in without developers noticing. Breakdowns in hub functionality generally occur when the hub is first set up, after a hub software upgrade, or after a hub server reboot. Below, we explore examples of problems seen on the hub and investigate, at a high level, how HUBcheck can be used to alert hub developers of the issues.

4.1 Hub Configuration Issues

There are two types of configurations the HUBzero Team seeks to support: a configuration for hubs managed internally by the group and a configuration for the open source release of the HUBzero Platform. Out of the box, the hub provides user friendly default configuration values for the open source release. Hubs managed internally use a slightly different configuration because the software often includes advanced features not available in the open source release, like upgraded middleware and web components. These differences can lead to misconfigured internally managed hubs. One example where support for these different configurations can be seen is in the `My Sessions` module, available on the user's Dashboard on the hub website.

The `My Sessions` module was designed to provide a way for users to manage their active tool session containers. When fully configured, the My Sessions module can also show a screenshot of the tool session container, provide a shortcut link to open the tool session container, and display the user's available disk space on the hub. Figure 4.1 compares a fully configured My Sessions module, on the left, with one whose features are disabled or not working properly, on the right.

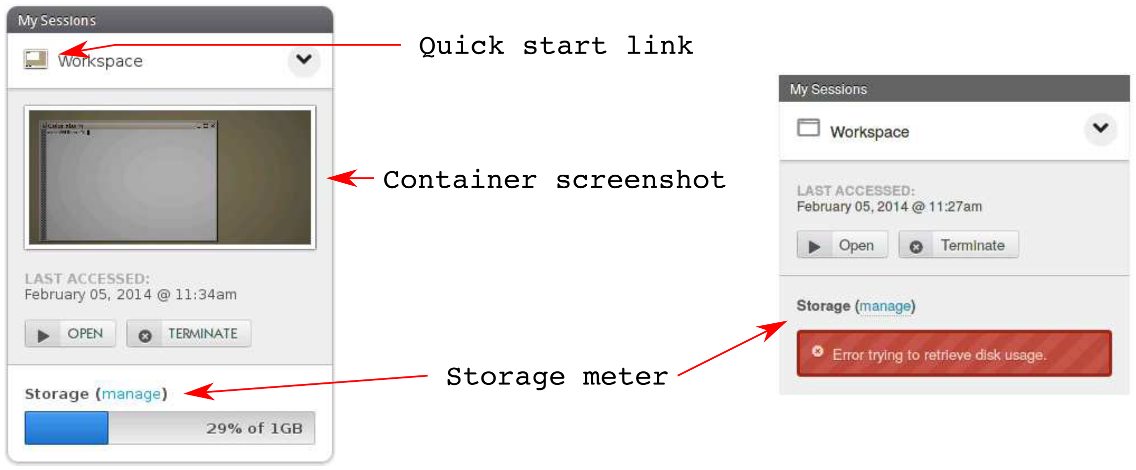


Fig. 4.1.: The `My Sessions` module can be configured to show screen shots of the tool session containers, provide short cut links to access the container, and display the user's available disk storage on the hub. The module on the left is fully configured showing the container screenshot, an enabled quick start link, and storage meter. The module on the right has some of these features disabled.

The My Sessions module can be configured to provide the user with a shortcut link to access active tool session containers and screen shots of applications running inside of an active tool session container. Capturing screen shots of the tool session container is a function performed by the middleware and is not available in the open source release prior to version 1.2.1. Because of this, the feature is turned off by default. Hosted hubs, managed by the HUBzero Team, run an advanced version of the middleware that supports this feature. When new hosted hubs are launched, turning this feature on is often missed.

Similarly, the availability of the disk usage and quota information depends upon the hub being configured to show the information and having the **telequotad** service running on the hub's fileserver. When the hub is not configured to show the disk usage, or the telequotad service is not running, users are shown messages like those in the image on the right side of Figure 4.1, explaining that the information is unavailable or simply stating the used disk space is **0% of 0GB** when the user has no quota set.

When a hub is first installed, there are many settings that can be adjusted to change the user experience. HUBcheck provides a library to help developers automate the validation of these settings through the hub's website, from the user's perspective. Using HUBcheck, developers can write automation scripts that can login to the hub website as a user, start a tool session container, and test if the My Sessions module is correctly showing screen shots and enabling short cut links.

Developers can also write HUBcheck based automation scripts to identify problems like the improper disk usage calculation mentioned earlier. To do this by hand a developer would:

1. login to the hub website as a user;
2. navigate to the user's Dashboard web page;
3. locate and read the disk storage string from the web page;
4. validate the string holds the proper format.

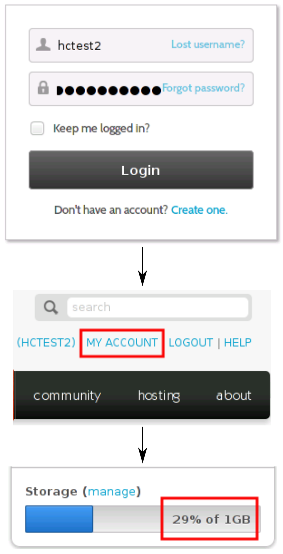


Fig. 4.2.: Accessing user's storage meter.

The HUBcheck script shown in Listing 4.1 follows the same steps. The format of the disk storage string should match the format similar to **X% of YGB**, where X is an integer, ranging from 0 to 100, describing the percentage of the user's available disk that has been used, and Y is a positive integer describing the amount of disk space available to the user, in gigabytes.

HUBcheck's web automation library simplifies common tasks like logging into the hub website and navigating to web pages. The library also provides abstractions of hub web pages, called page objects, so developers can reuse common blocks of code and take advantage of a standard library for locating elements on the web page.

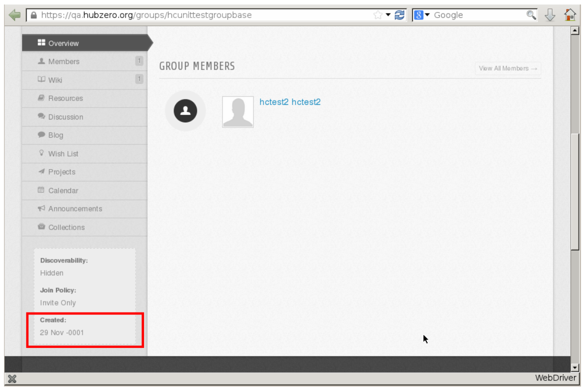
Listing 4.1: Checking user's storage meter, using a HUBcheck backed script

```
...
# launch the browser and navigate to hub website
hc.browser.get('https://hubzero.org')

# login to the hub website as a user
hc.utils.account.login_as(username,userpass)

# navigate to the user's Dashboard web page
po = hc.catalog.load_pageobject('GenericPage')
po.header.goto_myaccount()

# locate and read the disk storage string from the web page
po = hc.catalog.load_pageobject('MembersDashboardPage')
storage_amount = po.modules.my_sessions.storage.storage_meter()

# validate the string holds the proper format
assert storage_amount != '', 'invalid storage amount returned'
assert storage_amount != '0% of 0GB', 'user quotas not activated'
...
```

4.2 Hub Upgrade Issues

Errors also tend to arise on the hub after software upgrades. These types of errors can usually be traced back to configuration changes in upgraded hub modules, the release of errant software, or pre-existing errors on the upgraded hub that manifest themselves after the upgrade. Running HUBcheck after a hub has been upgraded can help identify errors related to hub upgrades before the user experiences them.

One example of a hub upgrade related error that HUBcheck was able to identify occurred in the hub's `Groups` component. The `Groups` component allows users to organize content and discussions within the hub community. The component provides discussion forums, wiki pages, blogs, project spaces and more. Users can create a group by filling out a web form on the hub website. Groups on the hub have



Discoverability:
Hidden

Join Policy:
Invite Only

Created:
29 Nov -0001

Fig. 4.3.: Sometimes errors show up after a hub upgrade like this one, where the create date of a new hub group was not being displayed correctly. Finding this type of bug is tedious for a human, but developers can use HUBcheck to write tests which verify that multistep processes, like creating a group, still produce expected results.

an overview web page that provides some properties of the group like group name, description, number of members, join policy and create date.

In this case, HUBcheck was run on a test hub after a software upgrade. A failing test alerted developers to an error in displaying the create date for newly created groups. With knowledge of the problem, hub developers were able to identify and fix the the errant code before it was propagated to more hubs, which would have resulted in additional cleanup work.

To find this type of error, a new group needed to be created on the hub, and its properties verified. The process of creating and verifying a new group on the hub can be tedious and error prone for a human, but with HUBcheck it can be done in a few lines of code in a testing script.

4.3 Hub Reboot Issues

When the servers hosting a hub are shut down and brought back up, it is easy for unexpected problems to arise. From hosts having trouble rebooting to remote filesystems not being properly mounted over the network, machine reboots are a time where errors can happen that leave the hub not living up to its fullest potential.

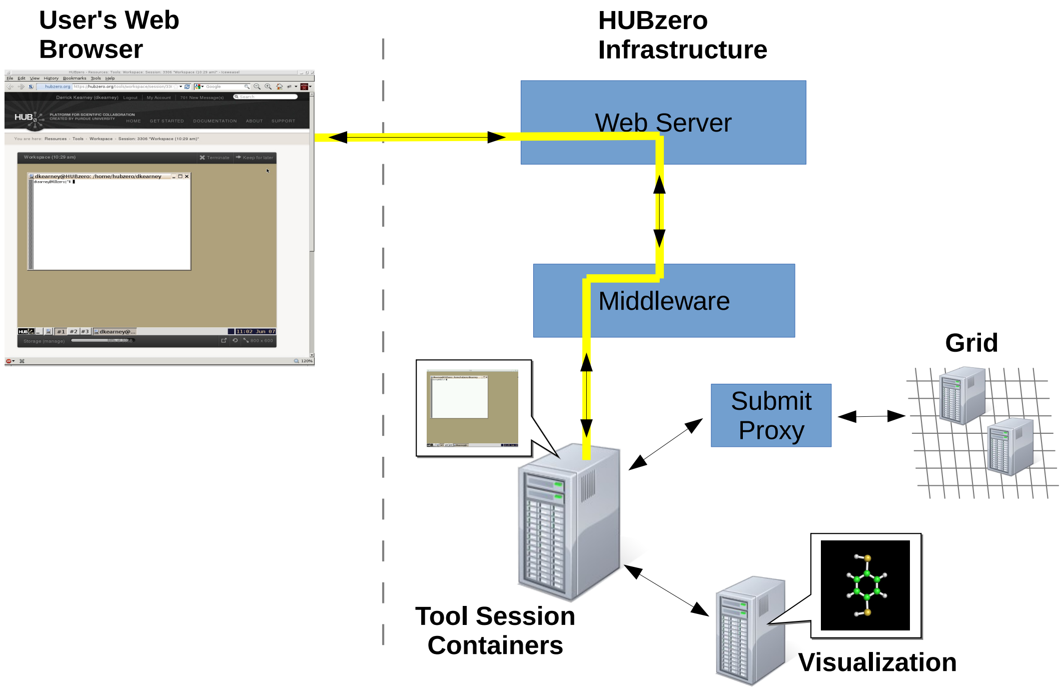


Fig. 4.4.: Simulation tool developers use a tool session container to develop and deploy their work on the hub. Checking that the containers are properly configured is time consuming for a human because of the layers of software between the web browser and the container's services like Submit and the Visualization servers.

Developers can exercise hub functions quickly by running HUBcheck after a reboot. HUBcheck provides the type of automation primitives that encourage developers to tackle the harder problems to automate, like checking that render servers are accepting connections from within a tool session container.

One of the key features of a hub is its ability to run interactive simulation tools that are displayed in the user's web browser. These simulation tools are run in an environment called a tool session container on an execution host that is a part of the hub. In their web browser, the user sees a display of the tool session container that has been projected to them, from the hub, using the VNC protocol. Hosting the interactive simulation tools on the hub allows the user to take advantage of many features that would not normally be found on their own system, like access to grid computing services and powerful rendering machines for interactive visualization.

The tool session container is a Debian Linux environment that supports the building of simulation tools by tool content developers and the execution of tools by hub users. Access to grid computing and render machines are services provided to the tool session containers. After a reboot of the hub these services should be restored, and if they are not, the simulation tools may not work correctly.

There are two ways to access a tool session container. The most frequently used way is through the web browser. Starting a tool on the hub gets the user access to a tool session container. This approach doesn't allow the user to automate interaction in a terminal with a shell. The second way is to connect over SSH, the secure shell protocol. All tool developers can access a tool session container in this way. This approach has the advantage that it gives access to a terminal with a shell, and shell automation tools like Expect [29] have existed since the mid 1990s.

HUBcheck takes advantage of this second approach to accessing a tool session container and provides a small, Expect-like, shell automation library. Using HUBcheck, hub developers can write automation scripts that enter a tool session container and examine the resources that are supposed to be available for tools to use, like access to the render servers. They can also write scripts to check if a render server is accepting connections from within the tool session container, examine the container firewall setup, installation of software such as the Rappture Toolkit, access to grid infrastructures through the use of the **submit** command, file transfer between the user's desktop and the user's hub account using the `sftp`, `filexfer`, or `webDAV` protocols, and simulation tool invocation through **invoke_app**, all under the same conditions a simulation tool would be making its request from.

4.4 Multimodal System Automation

HUBcheck's combination of web and shell automation libraries helps it provide developers with the unique ability to write scripts that capture the user experience of working in the dual environment system that is the hub. While a great deal of testing and automation can be performed by libraries that only access the website, or only access the tool session container, there exists a set of tasks whose operations span both the website and the tool session container, that no other single tool can automate. These are the cases of a growing area of interest within the hub, where information is passed from resources published on the website to tools running inside of the tool session container. In hub parlance, this is referred to as **parameter passing**.

Simulation tools run in a tool session container, either on the same host as the hub's web server or on a separate execution host. While this approach has its advantages with respect to deploying tools in a consistent environment – isolation between different users running tools and isolation between tools and the web server – there are also a number of disadvantages, one of which is the inability to easily pass parameters to the simulation tool before it has launched.

Launching a simulation tool on the hub involves coordination between both the hub website and hub middleware. It can be explained in five steps and are shown in Figure 4.5:

1. User click's link to launch tool from web page.
2. Web link calls PHP function which forwards request to middleware.
3. Middleware starts a new tool session container for the tool.
4. Middleware calls the tool's invoke script inside of the tool session container.
5. Invoke script execs command to start tool's graphical user interface.

There are a couple of ways to start the process to launch a simulation tool on the hub. The most obvious way is to use a web browser to navigate to the tool's *tool*

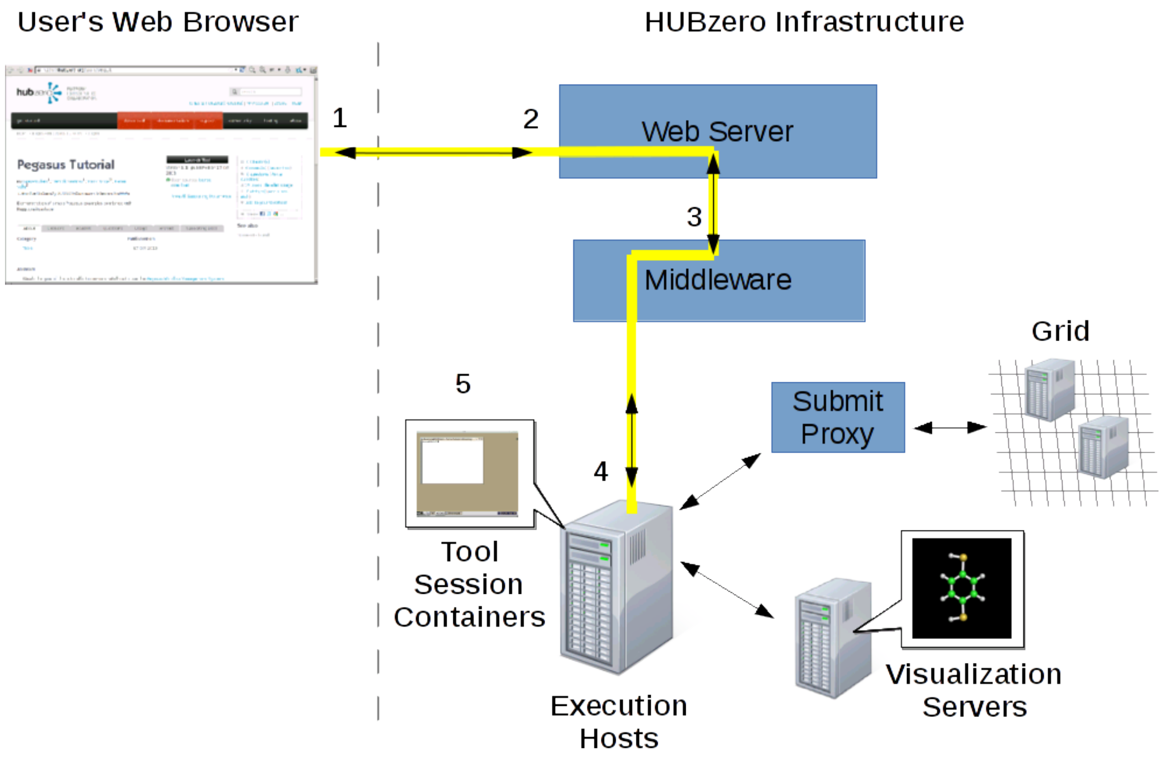


Fig. 4.5.: Simulation tools are started by user's clicking a link on the hub website. The link forwards the request to the middleware, which handles allocating a tool session container and calling the tool's invoke script. The invoke script sets up the environment for the tool to run in, and finally, launches the tool.

information page, a web page that describes what the tool does, lists its authors and funding sources, and includes a link to launch the tool. In the first step, the user navigates to the tool information page and clicks the link to launch the tool. Clicking the web link sends a request to the hub web server asking it to start the tool. The hub web server receives the request and calls upon the hub middleware to launch the new tool. In step three, the middleware supplies a tool session container to run the requested tool.

Tools published on the hub include an *invoke script* which contains all of the commands necessary to launch the simulation tool, including setting up system environment variables with prerequisite library and executable paths. In step four above, the middleware enters the tool session container as the user, and execs the tool's invoke script to start the graphical user interface. Lastly, the invoke script sets environment variables for the libraries needed by the tool and launches the tool.

Version 1.2 of the HUBzero software included new components that allowed users to interact with learning concepts and data through the use of simulation and modeling. Two examples of this include the Databases component and the Courses component. The Databases component allows users to create databases of information and construct views that help others understand their data. The Courses component allows teachers to manage an online class, hosted on the hub, that incorporates hub resources including simulation tools. With both of these components, developers may want to pass data that is stored on the hub website over to a tool running in a tool session container.

To address this need, a new algorithm for allowing parameters to be passed from the website into a simulation tool running in a tool session container was created. The algorithm accepts a limited number of data types (file names, directory names, and integers) and encodes the parameters into the URL used to launch the tool. The parameters are passed through the hub web server and middleware, which have the opportunity to check them for validity, and into the tool session container. Once inside of the tool session container, they are stored in a file, and the tool developer is responsible for parsing them out of the file, either through the tool's invoke script, or in the simulation tool itself.

Whether it is data from a database being fed into a cancer prediction model, or example parameters for simulating a circuit from a class, passing data between the website and the tool session container involves many layers of software that have the opportunity to manipulate or lose the data. Passing data is difficult to do and tedious to test. HUBcheck provides the automation primitives necessary to ease the task of writing scripts that can interact with the website and tool session container in the same script. In the case of parameter passing, hub developers were able to use HUBcheck's web and shell automation libraries to build a test suite to exercise passing parameters to a simulation tool by crafting both valid and invalid URLs. As a part of the test suite, numerous simulation tools were installed on the hub and URLs were generated to match the requirements and restrictions of the parameter passing algorithm. After launching tools with the specialized URLs, the HUBcheck based scripts accessed the tool session container running the simulation tool and verified that the parameters were passed through the tool invocation labyrinth (which includes the web browser, web server, middleware, and tool session container), through the tool's invoke script, and to the tool where they could be processed.

5. EXTERNAL LIBRARIES

HUBcheck builds upon several external libraries to provide web browser and shell based automation. To support web automation, HUBcheck uses the Selenium WebDriver API [30] and the BrowserMob Proxy [31]. The WebDriver API is used to control the web browser and is provided by the Selenium project along with bindings for several programming languages. The BrowserMob Proxy is an open source web proxy, maintained by Patrick Lightbody, which can be used to watch and manipulate network traffic. To support shell automation, HUBcheck builds upon Paramiko [32], a Python implementation of the SSH version 2 protocol. Below we'll learn more about the role each of these libraries plays.

5.1 Selenium WebDriver

Selenium is an open source project with a suite of tools used to automate web browsers across many platforms. Selenium implements the WebDriver API as a part of the Selenium WebDriver library. The WebDriver API provides an object-oriented programming interface to communicate with and control a web browser, performing the same actions a person would when interacting with a web page. Through the WebDriver API, developers can launch a web browser, locate elements in web pages, and perform actions on elements such as typing into them or clicking on them using the mouse. The following subsections show examples of how Selenium WebDriver can be used to control a Firefox web browser using the Python programming language.

5.1.1 Launching a Web Browser

The Selenium WebDriver Python bindings provide the `webdriver` module that holds classes to represent the different types of browsers that can be launched, including Firefox, Chrome, Safari, Opera, and PhantomJS. Browsers can be launched locally on the same machine running the automation program, or on a remote host that is running a Selenium Server. Each browser has a special plugin which translates the WebDriver requests, made from Selenium, to the browser's native automation API.

Launching a web browser using Selenium is as simple as instantiating a new `webdriver` object for the browser the user wants to launch. With no initialization arguments, the user will be provided with a new browser they can control by calling member functions of the returned object as demonstrated in Listing 5.1.

Listing 5.1: Launching a locally hosted Firefox web browser using Selenium WebDriver's Python API

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get('https://hubzero.org')
```

For some browsers, more fine grained web browser controls are exposed to the developer through access to the browser's profile. Not all browsers support the profile concept. Firefox allows users to load a pre-configured profile or create one on the fly.

Listing 5.2: Adjusting the preferences in the Firefox browser.

```
from selenium import webdriver

profile = webdriver.FirefoxProfile()
profile.set_preference('browser.startup.page',0)
profile.set_preference('app.update.enabled','False')
profile.add_extension('firebug-1.11.4.xpi')
profile.set_preference('extensions.firebug.currentVersion','1.11.4')

browser = webdriver.Firefox(firefox_profile=profile)
browser.get('https://hubzero.org')
```

Listing 5.2 demonstrates setting up a Firefox browser profile that disables the startup page, disables automatic updates to the browser, and installs the Firebug browser extension. The FirefoxProfile class allows developers to set preferences and add extensions to the Firefox browser. Alternatively, users can take advantage of the class's `profile_directory` argument to load a pre-configured browser profile.

The `webdriver.Firefox` class is used to launch the web browser. If a custom profile was created it can be provided to the class and a web browser based on those settings will be started. The object returned, shown in line 9 of Listing 5.2, is stored in the variable `browser`, and is used in automation scripts as the object reference for the web browser. All commands to the web browser, such as navigation and locating web elements in the HTML DOM (Document Object Model), are performed on the `browser` variable using the `get()` and `find_element()` family of methods.

5.1.2 Locating Elements on the Web Page

To perform actions on elements of a web page, Selenium must first be able to locate the elements. Web element *locators* are used by Selenium commands to identify elements on a web page. There are several different strategies for locating web page elements listed in Table 5.1. Two of the most popular strategies are XPath expressions and CSS selectors.

There is typically overlap in how the different locator strategies can be used to locate elements on a web page. It is not uncommon for a single web element to be identifiable by two or three of the locator strategies, but the key to building robust automation scripts is to choose the most robust locator that will withstand updates to the web page layout.

Table 5.1: Selenium element locator methods provide a variety of ways to find elements on a web page.

Locator Strategy	Description	Example Use
id	Search for a web element with an id attribute matching the argument.	id=username
name	Search for a web element with a name attribute matching the argument.	name=username
XPath	Search for a web element using an XPath expression.	xpath=//input[@id='username']
link	Search for a link (anchor web element) with text matching the provided pattern.	link=link text
CSS	Search for a web element using a CSS style selector.	css=input#username

Listing 5.3: HTML for username field

```
<label>
    Username:
    <input id="username" class="
        inputbox" type="text" name=
        "username">
<\label>
```

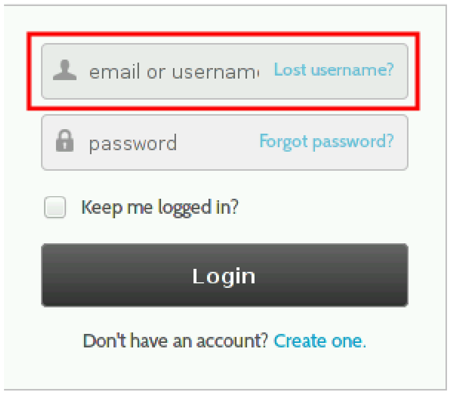


Fig. 5.1.: Hub login form.

Take, for example, the hub login form shown in Figure 5.1. The form contains a number of web elements we would like to interact with, the most important of which are the username field, password field, and login button. Listing 5.3 shows the HTML representation of the username element in the login form.

Several of the locating strategies in Selenium can be used to identify the element. The examples in Table 5.1 show how this would be done. Among the available strategies, the ones involving the `id` attribute are generally the most robust because HTML requires that the id attribute be unique for all elements. This ensures that if the id attribute exists for an element in the HTML DOM, then it should only exist once, reducing the number of false positives when locating elements.

The locating strategies map directly to the Selenium WebDriver API functions used to locate elements on the web page from a program. For the Python bindings, these are a part of the `find_element_by_*()` family of methods shown in Listing 5.4 Any one of these methods can be used to locate elements on a web page.

Listing 5.4: The find_elements_by_*() functions locate web elements in Python.

```
# locating elements by the id attribute
element = browser.find_element_by_id('username')

# locating elements by name attribute
element = browser.find_element_by_name('username')

# locating elements by tag name
element = browser.find_element_by_tag_name('input')

# locating elements by XPath
element = browser.find_element_by_xpath("//input[@id='username']")

# locating elements by CSS Selector
element = browser.find_element_by_css_selector('input#username')
```

5.1.3 Performing Actions on Web Elements

Once an element has been located, an action can be performed on the element. Common actions include clicking on the element, sending key strokes to the element, reading the properties of the element, and checking if the element is displayed. Listing 5.5 demonstrates performing actions on the username field of the login web page. In line 2, the username field is located using a CSS selector element locating strategy. Lines 8 - 10 show that elements can be brought into focus on the web page with the `click()` method, have its value erased using the `clear()` method, and a new value assigned using the `send_keys()` method. By the end of this program, the username field on the login page would be populated with the name “hctest”.

Listing 5.5: Filling in the username field on the login form

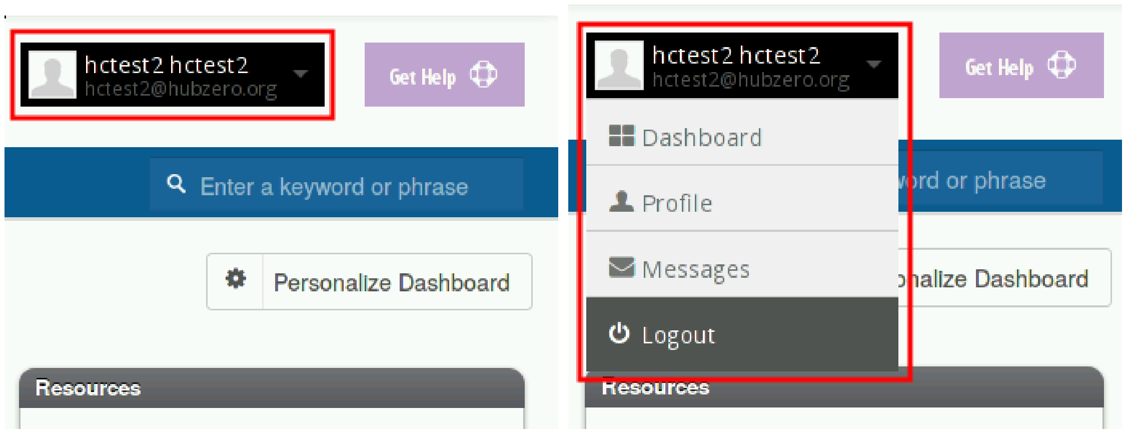
```
# locate the username field by CSS Selector
element = browser.find_element_by_css('input#username')

# perform actions on the element:
# click the field to set focus
# clear any previous value from the field
# send key strokes to the field to fill in the username
element.click()
element.clear()
element.send_keys('hctest')
```

5.1.4 Performing Mouse Actions on Web Elements

In addition to being able to perform normal actions on elements, Selenium WebDriver allows automation developers to perform mouse actions on elements with a feature called ActionChains. ActionChains are useful for automating tasks where a mouse movement is needed to trigger a property change in an element on the web page. A common examples of this include JavaScript based menus that appear on the screen when the mouse hovers over an element on the web page, as shown in Figure 5.2a.

Selenium tries hard to only allow interaction with visible page elements that a user would be able to interact with. If an element is present on the web page and is available in the HTML DOM that the browser loaded, but is not visible to the user, Selenium will not allow the automation code to perform actions on it. Properties, like the text of the element or the attributes of its HTML, can still be read, but clicking on the element or sending key strokes to the element will fail.

To emulate the movement of the mouse, the automation developer can use Selenium WebDriver’s ActionChains. ActionChains allow the automation developer to specify elements on the web page to perform mouse actions on. ActionChains support hovering the mouse over elements, single clicking elements, double clicking elements, context (right) clicking elements, clicking and dragging elements, and pressing a number of meta (ctrl, alt, shift) key combinations in coordination with a mouse action.



(a) User account menu closed. (b) User account menu open.

Fig. 5.2.: Selenium WebDriver provides ActionChains to automate performing mouse actions on elements of a web page. ActionChains can be used for things like right or left clicking on an element, drag and drop, and hovering the mouse over an element.

On the hub, performing mouse actions is handy when trying to interact with the user account menu, which requires the user to hover the mouse over the menu element, shown in Figure 5.2a, in order to expose account navigation options that lead to the user's Dashboard, Profile, Messages, or log the user out of the website. To do this programmatically using the Python bindings, an automation script would use the `action_chains` module.

Listing 5.6: Activating a JavaScript menu using ActionChains.

```
 1 # load the ActionChains class
 2 from selenium.webdriver.common.action_chains import ActionChains
 3
 4 # locate the menu and logout elements by CSS Selector
 5 menu_element = browser.find_element_by_css('#account')
 6 logout_element = browser.find_element_by_css('#account-logout')
 7
 8 # build the ActionChains object to perform a mouse action:
 9 # move the mouse over the menu to activate the JavaScript menu
10 # move the mouse to the logout list item in the menu
11 # single click the logout menu item
12 actionProvider = ActionChains(browser)
13 actionProvider.move_to_element(menu_element)
14 actionProvider.move_to_element(logout_element)
15 actionProvider.click()
16 actionProvider.perform()
```

Developers use the `ActionChains` class to build (chain) a list of actions together and send them to the web browser to be performed as if they came from the computer's mouse. Listing 5.6 demonstrates this by providing a solution to the problem of hovering the mouse over the menu in Figure 5.2a, exposing the account navigation options shown in Figure 5.2b. More specifically, the script focuses on clicking the `Logout` link in the menu.

As with most Selenium WebDriver related scripting tasks, the first step is to locate the web elements items the script will be interacting with in the HTML DOM. Lines 5 and 6 of Listing 5.6 are responsible for locating the menu element and the logout link on the web page. Next, in lines 12 through 15, the script builds up a list of commands that should be performed by the mouse. Each line in the script adds another command that should be performed to the list stored in the variable `actionProvider`. To get to the logout link, the mouse must first move to the menu element, line 13, then move to the logout element, line 14, and lastly send a click signal to press the logout link, line 15. The commands are stored until the developer asks for them to be performed, shown in line 16 of the script.

5.1.5 Waiting for Web Elements

Asynchronous JavaScript and XML (AJAX) is a programming technique that allows web applications, running in the user's browser, to communicate with the web server and dynamically change the state of the web application without reloading the web page. The use of AJAX on web pages poses a problem for some web automation software that evaluates static HTML DOMs without evaluating the JavaScript that accompanies it. After the browser makes a request for a web page, the web server sends back a stream of HTML that can have JavaScript embedded within it. The web browser is responsible for reading the HTML and evaluating the JavaScript inside, which may request additional HTML from the web server.

Selenium WebDriver uses the HTML and JavaScript engines inside of the external web browser it is controlling to build the HTML DOM that represents the web page the web browser loaded. This allows Selenium WebDriver to leverage the same web browser to interpret and evaluate the HTML and JavaScript that a user would. This also adds a dimension of difficulty in locating web elements whose presence or visibility on the web page is influenced by JavaScript. The dynamic nature of JavaScript provides less structure to what it means for a web page to have finished being 'loaded'. Loading up multiple pieces of a web page may be delayed, for example, by a slow network. Selenium WebDriver provides two types of waiting strategies, *Implicit Waits* and *Explicit Waits*, to work around these unexpected delays and deal with locating elements that may not be immediately available.

Implicit Waits

In Selenium WebDriver, the default behavior is to search once for a locator in the HTML DOM. If the locator cannot be found, a `NoSuchElementException` is raised. Implicit Waits are a way to repeatedly search for a web element in the HTML DOM, while blocking other commands from continuing, for a set amount of time before raising the `NoSuchElementException`.

Implicit waits can be setup once and exist for the life of the webdriver object. Setting up an implicit wait affects the whole family of `find_element*()` based methods, which will repeatedly search for the element until either the element is found or the timeout limit is hit.

Explicit Waits

Explicit Waits consider the more general case where the automation developer wants to wait for a condition to be met. Without explicit waits, many people are tempted to check for the condition to be met while inside a `for` loop, adding a call to the programming language's `sleep()` function to space out the checks. This approach should be avoided in favor of using the `WebdriverWait` class provided by the Selenium libraries.

5.1.6 The Page Object Design Pattern

The Page Object design pattern provides a layer of abstraction between a web page and automation scripts. It represents the services offered on a web page or a portion of a web page. By writing automation scripts that interact with the page object, developers can significantly reduce the number of repeated lines of code in their automation scripts, abstracting away many of the common features of the code into a single class that needs to be updated as the web page interface changes.

Novice developers learning to use Selenium for their web automation may start by using the Selenium IDE, a graphical user interface product of the Selenium project that records the actions of the user and stores them using an internal representation called *Selenese*. Selenium IDE also has the ability to convert Selenese to a number of programming languages. The IDE allows developers to quickly create automation scripts, but it suffers from the same afflictions as manually writing automation scripts using the procedural programming paradigm. They both result in brittle scripts that break easily and are painful to fix because of a lack of encapsulation. With a little organization and refactoring, these brittle scripts can be turned into robust scripts that require less code to write and are easier to understand.

Consider the automation script in Listing 5.7, which navigates the web browser to the `hubzero.org` web page, clicks the link to login, then fills in the username field, fills in the password field, and clicks the submit button.

Listing 5.7: Simple hub login automation script.

```
1  from selenium import webdriver
2
3  # setup automation script constants
4  base_url = "http://hubzero.org/"
5
6  # start the browser
7  driver = webdriver.Firefox()
8
9  # navigate to the login page
10 driver.get(base_url)
11 driver.find_element_by_id("account-login").click()
12
13 # perform the login action
14 driver.find_element_by_css_selector("#username").clear()
15 driver.find_element_by_css_selector("#username").send_keys("abc")
16 driver.find_element_by_css_selector("#passwd").clear()
17 driver.find_element_by_css_selector("#passwd").send_keys("123")
18 driver.find_element_by_css_selector("[name='Submit']").click()
```

Immediately looking at the code, a couple of patterns are apparent. The first pattern involves repeated searches for commonly used fields. This is true of the field with id `username`, which is searched for in line 14 to clear it and again in line 15 to send it a value, and also for the the password field with id `passwd` for the same reasons in lines 16 and 17. To clean up this code, it could be rewritten to save the result of the first search to a variable and call the `clear()` and `send_keys()` methods on that variable.

The second pattern involves the repeated steps used to populate a text field. Every time a text field is populated, the script first searches for the element, then clears the element of any previous value, and lastly, sends some keys to the element. These actions could be combined into a function, as shown in Listing 5.8 to help reduce the amount of repeated code.

Listing 5.8: populate_input function included in utils.py

```python
def populate_input(driver,loc,text):
  """type text into the element located by loc"""

  e = driver.find_element_by_css_selector(loc)
  e.clear()
  e.send_keys(text)
```

An updated version of the automation script from Listing 5.7 that addresses the first two patterns, including a function named `populate_input()` which finds an element and types text into it, would look like this:

Listing 5.9: Hub login script with repeated patterns abstracted away

```python
from selenium import webdriver
from utils import populate_input

# setup automation script constants
base_url = "http://hubzero.org/"

# start the browser
driver = webdriver.Firefox()

# navigate to the login page
driver.get(base_url)
driver.find_element_by_css_selector("#account-login").click()

# perform the login action
populate_input(driver,"#username","abc")
populate_input(driver,"#passwd","123")
driver.find_element_by_css_selector("[name='Submit']").click()
```

At this point the automation script in Listing 5.9 is looking pretty good. We were able to abstract out most of the repeated code into a function, `populate_input()`, that we can pass arguments to and let it take care of locating and writing text to elements of the HTML DOM. The act of logging into the website has been condensed down to lines 15-17 in the automation script, but this version of the automation script could still be considered brittle.

Many web automation script errors are caused by failures to locate elements in the HTML DOM. This could be the result of poor element locator strategy or just a new site design being implemented. If multiple automation scripts exist and they all repeat lines 15-17 from Listing 5.9, then they all need to be changed to reflect the new design. The more scripts there are, the more painful the processes of updating all of them is.

An alternative approach involves identifying common pieces of web pages that can be interacted with and representing them as objects in code. The methods of these objects are services provided by the web page. When the automation script navigates to a web page, it would instantiate a page object, an object that represents the web page, and call the page object's methods to perform actions on that web page.

Applying the Page Object design pattern to the automation script in Listing 5.9, we first identify the web pages being visited. The first web page being visited is the hub's index page. On the index page, the automation script locates and clicks the login link. The first page object we create should represent the index page, and it should provide the service of clicking the login link as one of its methods. Listing 5.10 shows a page object representing a generic web page where the user is logged out.

Listing 5.10: A generic page object, providing the login navigation service.

```
class GenericLoggedOutPage(object):

  def __init__(self,driver):
    self.driver = driver

  def goto_login(self):
    """navigate to the login page"""

    self.driver.find_element_by_css_selector('#account-login').click()
```

In Listing 5.10, `GenericLoggedOutPage` objects are initialized with a handle to the web browser through the `driver` variable. The class has a method named `goto_login()` which provides an interface to the service of navigating to the login web page by locating and clicking the login link.

After clicking the login link, the automation script in Listing 5.9 fills in the login page web form with a username and password, then clicks the submit button to complete the login action. These commands can be grouped into another page object that represents the login web page. In Listing 5.11, the `LoginPage` page object provides the `login_as()` method to represent the service of filling in and submitting the login form.

Listing 5.11: The Login page object represents the services provided by the login web page

```
from utils import populate_input

class LoginPage(object):

  def __init__(self,driver,loctype):
    self.driver = driver

  def login_as(self,username,password):
    """login a user by typing the username and password
       into the login form, and pressing the submit button
    """

    populate_input(driver,'#username',username)
    populate_input(driver,'#passwd',password)
    self.driver.find_element_by_css_selector("[name='Submit']").click()
```

Listing 5.12 shows an updated version of the automation script which incorporates the new page objects. The updated automation script puts more focus on the current web page and delegates the steps to perform the actions to the page objects. The page objects can be used in multiple automation scripts which promotes fewer lines of repeated code, and cleaner looking, more readable automation scripts. Additionally, all of the actions for a particular web page are centralized in the page object. When the layout or locators for a web page change, only the page object needs to be updated, provided the services of the web page don't change. This type of encapsulation is one of the biggest advantages of the Page Object design pattern.

Listing 5.12: Revised automation script, using Page Objects to perform actions.

```
from selenium import webdriver
from po_login import LoginPage
from po_generic_logged_out import GenericLoggedOutPage

# setup automation script constants
base_url = "http://hubzero.org/"
username = "abc"
password = "123"

# start the browser
driver = webdriver.Firefox()
driver.get(base_url)

# navigate to the login page
po = GenericLoggedOutPage(driver)
po.goto_login()

# perform the login action
po = LoginPage(driver)
po.login_as(username,password)
```

5.2 BrowserMob Proxy

Web browser automation tools provide functions to perform the actions a user would manually perform inside of a web browser. Beyond just automating the web browser, writing programs to mimic the user experience sometimes requires additional information about the result of a web request that is not always apparent based on the elements available on the web page. Some of this information can be gathered by using a web proxy in front of the browser.

The purpose of the web proxy is to monitor and manipulate interactions between the browser and web server. Because of its position as a man-in-the-middle, the web proxy can record information about the requests from the web browser and responses by the web server. The standard format used to store this type of information is the HTTP Archive (HAR) format [33], now in version 1.2.

The BrowserMob Proxy [31] is a web proxy that captures the interactions of the web browser and the web server and reports it back to the user in the HAR format. A separate project supplies Python language bindings for communicating with the proxy server and starting new clients. The BrowserMob Proxy fits in well with web browsers being controlled by Selenium WebDriver.

The BrowserMob Proxy uses a single server to field requests for starting and configuring proxies. The server is started once and is responsible for managing the proxies. Separate proxy instances, referred to as clients, are started for each browser. Web requests are made by the browser and funneled through the proxy client to the web server.

The BrowserMob Proxy server provides a RESTful [34] API that allows users to programmatically control the server and proxy clients via simple HTTP requests. Since all of the server commands are HTTP requests, they can be made using a program like **curl** [35], a command line utility for transferring data with URL syntax. The Python language bindings [36], which were developed by a third party, provide similar functionality. Listing 5.13 demonstrates how to start the proxy and attach it to a Selenium WebDriver controlled web browser.

Listing 5.13: Starting the BrowserMob Proxy server and client

```
from selenium import webdriver
import browsermobproxy

bmp_path = '/usr/local/bin/browsermob-proxy'

proxy_server = browsermobproxy.Server(bmp_path,{'port':9090})
proxy_server.start()

# start up a proxy client
proxy_client = proxy_server.create_proxy()

# block any requests to facebook
proxy_client.blacklist('http(s)?://.*facebook\\.com/?.*',200)

# launch a browser, setting the proxy, load a web page
browser = webdriver.Firefox(proxy=proxy_client)
browser.get('https://hubzero.org')

# get the har of the last request from the browser
har = proxy_client.har

# close down the proxy server and client
proxy_client.close()
proxy_server.stop()
```

5.3 Paramiko

Paramiko [32] is a Python based implementation of the SSH version 2 protocol for making secure connections between machines. Using Paramiko, users can programmatically open secure channels for access to services on remote machines including shells and SFTP.

Listing 5.14: Starting a SSH and SFTP connection using the Paramiko library.

```python
import paramiko

# setup the SSH client and authenticate
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.load_system_host_keys()
client.connect(
    hostname='hubzero.org',
    port=22,
    username='user1',
    password='user1password')

# execute a single command
stdin, stdout, stderr = client.exec_command('ls')

# invoke a new shell and execute multiple commands
channel = client.invoke_shell(width=1000,height=1000)

# start up an SFTP channel
transport = client.get_transport()
sftp = paramiko.SFTPClient.from_transport(transport)
```

6. HUBCHECK

6.1 What is HUBcheck?

HUBcheck is a Python based library that allows users to build and run automation tools that involve HUBzero software. While the most obvious use of the HUBcheck library is for testing the hub website and tool session containers, the software is designed for the more general purposes of automating tasks that involve a web browser or an SSH shell. In the past, the HUBcheck library has been used to enroll students into courses on the hub, submitting support tickets, and performing hub maintenance tasks.

The goal of the HUBcheck library is to provide interfaces that interact with HUBzero products in the same way a user would. In essence, HUBcheck mimics a user's experience, either on the command line, or in the web browser. Using the HUBcheck library, automation tasks can be written at a higher level, abstracting away many of the differences between the hubs hosted by the HUBzero Team, whether they are site design differences or HUBzero software version differences.

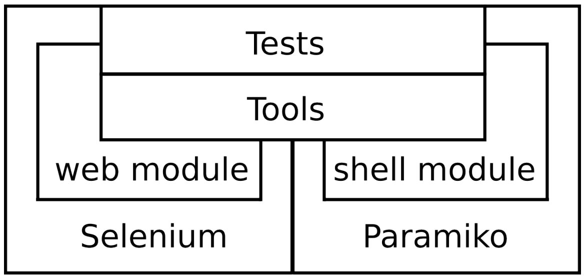


Fig. 6.1.: The HUBcheck library builds on top of the Selenium and Paramiko libraries.

The HUBcheck library uses pseudo terminals and web browsers to perform tasks. Pseudo terminal communication is managed by the `subprocess` and `paramiko` modules from the Python programming language, while web browser control is routed through Selenium WebDriver's Python bindings. The `hubcheck` package provides web modules and shell modules to perform many of the redundant tasks required to setup the resources needed to perform automated tasks.

6.2 HUBcheck Web Modules

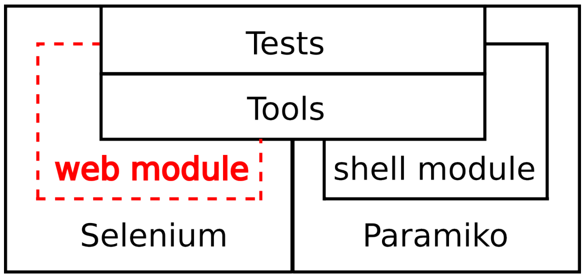


Fig. 6.2.: The HUBcheck library web module can be used to launch browsers and interact with web pages.

HUBcheck relies upon the web browser automation of Selenium WebDriver to manage tasks being performed on hub websites. Selenium WebDriver provides an abstract way of controlling a number of different web browsers including Firefox, Chrome, IE, Safari, and Opera. The most recent releases of HUBcheck support instantiating a Firefox web browser using the `Firefox` class. Once a web browser has been launched, users can begin to interact with web pages using HUBcheck's page objects, which provide common functionality for pieces of web pages.

Consider the use case where an automation script needs to login to the hub website as a user. In order to complete this task, the script must first launch a web browser that can be automated. Next the script needs to tell the web browser to navigate to the hub's login page. Once on the login page, the script has to fill in the username and password fields, and press the login button. Finally, the script should check that the login was successful. Listing 5.7 laid out a fragile way to do this with hard coded web element locators that were embedded inside of the automation script. This made the script hard to read and difficult to update. An alternative way to solve the problem, using the HUBcheck library, is shown in Listing 6.1.

Listing 6.1: Login to the hubzero.org website using the HUBcheck library

```
1  import hubcheck
2
3  username = 'user1'
4  password = 'pass1'
5
6  hc = hubcheck.Hubcheck(hostname='hubzero.org',
7                         locators='hubzero',
8                         browsertype='Firefox')
9
10 # setup a web browser
11 hc.browser.get('https://hubzero.org')
12
13 # login to the website
14 po = hc.catalog.load_pageobject('LoginPage')
15 po.goto_page()
16 po.login_as(username,password)
17
18 # check that the user is logged in
19 po = hc.catalog.load_pageobject('GenericPage')
20 assert po.is_logged_in() is True
```

Listing 6.1 contains no web element locators directly in the automation script. Instead, it takes advantage of the page objects that are a part of the HUBcheck library, which makes for a simplified script that is easy to read and understand. Listing 6.1 starts, on line 6, by creating a `Hubcheck` object. The `Hubcheck` object manages both the web browser and the page objects used in the script. Next, on line 11, the script uses the `Hubcheck` object to launch the web browser and navigate it to the hub's website. To get to the hub's login page, we load the `LoginPage` page object on line 14 and use its `goto_page()` method to handle the navigation. Once on the login page, we employ the page object's `login_as()` method to perform the login service of filling in the username and password fields, and clicking the login button. Lastly, the script checks that the user was properly logged in by calling the `is_logged_in()` method available in all HUBcheck page objects.

The sections below describe the details of opening a web browser, navigating the hub, and using the services available through HUBcheck page objects.

6.2.1 Configuring HUBcheck

HUBcheck supports multiple versions of the HUBzero software, so it is important that HUBcheck is configured properly before starting the web automation. The most basic configuration tells HUBcheck the type of web element locators to use and the URL of the hub where automation will occur. These two pieces of information can be fed into the `Hubcheck` object as parameters during initialization.

Listing 6.2: Configuring a Hubcheck object

```
hc = hubcheck.Hubcheck(hostname='hubzero.org',
                       locators='hubzero',
                       browsertype='Firefox')
```

Listing 6.2 demonstrates how to create and configure a `Hubcheck` object. The object allows developers to configure the hostname of the hub under automation, the page objects that should be used during automation, and the web browser to automate. The HUBcheck library has page objects and web element locators from 19 of the hubs managed by the HUBzero Team, as well as a few of the latest open source releases. Popular locators to use include `hubzero`, for the hubzero.org website, and `osr_1_3_0`, for the open source release available at the time of writing.

The `Hubcheck` object is a thin layer over three much more powerful objects, the browser object, the catalog object and the utils object. The next few sections discuss how these three objects are used while building HUBcheck based web automation tools.

6.2.2 Launching a Browser With the Browser Object

HUBcheck provides classes, representing each of the supported web browsers, to make launching a web browser simple and fast. The browser classes build upon Selenium WebDriver browser objects, setting up preferences, loading browser extensions helpful for debugging, and attaching the browser to a proxy for HTTP response inspection. When a `Hubcheck` object is created, it includes a browser object that can be used to launch a web browser by calling the `get()` method.

Listing 6.3: Launching a Firefox web browser using the Hubcheck object

```
hc.browser.get('https://hubzero.org')
```

The type of browser launched is controlled by the `browsertype` argument of the `Hubcheck` object's initialization. If no browsertype is set, the object defaults to launching a Firefox web browser.

Behind the scenes, the `Hubcheck` object is calling HUBcheck's `Firefox` class. The `Firefox` class is responsible for setting up the browser profile, attaching the browser to a web proxy, and launching the browser. The same class could be called, directly by the developer, to get a stand-alone browser object as shown in Listing 6.4.

Listing 6.4: Launching a stand-alone Firefox web browser using HUBcheck's `Firefox` class

```
browser = hubcheck.browser.Firefox()
browser.get('https://hubzero.org')
```

6.2.3 Navigating the Hub With the Catalog Object

The HUBcheck library provides page objects that express the services available on hub web pages. Automation developers can use these page objects to ease navigation through the hub website, perform tasks on hub web pages, or to build new page objects using HUBcheck's widgets.

The HUBcheck library's page objects are contained within Python modules. Additionally, each hub has a module defining which page objects work with the hub. These modules are accessible through the `hubcheck` object's `catalog` attribute, which is an instance of the `PageObjectCatalog` class. Through the `catalog` attribute's `load_pageobject()` method, users instantiate new page objects appropriate for the hub being automated.

Listing 6.5: Loading a HUBcheck page object for hub navigation

```
1 # load the LoginPage page object
2 po = hc.catalog.load_pageobject('LoginPage')
3 po.goto_page()
4 po.login_as('testuser','pass123')
```

Listing 6.5 shows an example of loading the `LoginPage` page object, for the login web page, and assigning it to the variable `po` in line 2. The script goes on to call the page object's `goto_page()` method, which is a general page object service for navigation, and the `login_as()` method, which represent a service available on the login web page. The login web page offers several other services like navigating to the username remind page, navigating to the password reset page, navigating to the hub registration page, and submitting a support ticket. All of these services are available through methods of the LoginPage page object.

6.2.4 Performing Common Tasks With the Utils Object

Automation scripts often contain repeated bits of code that perform common tasks like logging into a website or uploading a resource. Many times the code required to perform these tasks is copied between scripts. For large tasks, copying code between scripts reduces code maintainability. The HUBcheck library provides developers with the `utils` object which holds functions for commonly performed tasks related to user accounts, support tickets, and tool resource contribution.

Listing 6.6: Common hub tasks are made easier using the methods from the utils object

```
import hubcheck

username = 'user1'
password = 'pass1'

hc = hubcheck.Hubcheck(hostname='hubzero.org',
                       locators='hubzero',
                       browsertype='Firefox')

# setup a web browser
hc.browser.get('https://hubzero.org')

# login to the website
hc.utils.account.login_as(username,password)

# check that the user is logged in
po = hc.catalog.load_pageobject('GenericPage')
assert po.is_logged_in() is True
```

Previously, Listing 6.1 demonstrated using the `LoginPage` page object and its `login_as()` method to fill in and submit the hub login form. Logging into a hub website is such a frequently performed task, it was included as a part of the `utils` class. In Listing 6.6, the code that used to load the `LoginPage` page object, navigate to the login web page, and fill in the login form has been replaced with a call to `hc.utils.account.login_as()`.

6.3 HUBcheck Shell Modules

The HUBcheck library also supports automation of shell utilities that run over Secure SHell Version 2 (SSHv2), the protocol used to create encrypted channels to services hosted on remote machines. The HUBcheck library allows developers to quickly open interactive shells and **sftp** sessions for accessing remote hosts, including the tool session container on the hub.

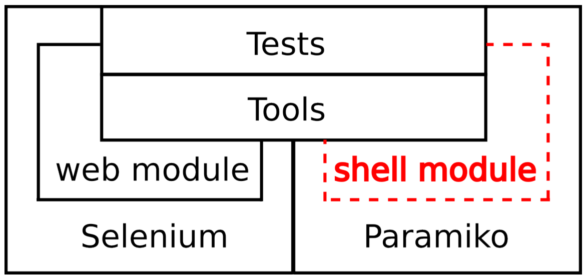


Fig. 6.3.: The HUBcheck library shell module can be used to SSH into systems and interact through the command line.

6.3.1 Starting a Remote SSH Session

The `hubcheck.SSHClient` class can be used to start an SSH session. The class allows for two ways to login to the remote host, either by providing a username and password or by providing a username and key filename, where the key filename is a file, or list of filenames, of private keys for SSH authentication.

Listing 6.7: Connecting to a remote host over SSH

```
import hubcheck

sh = hubcheck.SSHClient(host='hubzero.org',
                        username='testuser',
                        password='pass123')
```

Upon successful authentication, an `SSHShell` object is returned. The `SSHShell` object can be used to interact with the shell in a manner similar to Expect by using calls to `send()` and `expect()` methods. The `send()` method is used to run commands over the remote channel. It accepts a single parameter, the command to be run. The `expect()` method is used to retrieve output from the remote channel's buffer. It accepts a list of patterns and tries to match each pattern to the data in the channel's buffer. If it finds a match, the matching data is stored and the method returns the index of the pattern that matched the data. Listing 6.8 shows examples of using the `send()` and `expect()` methods.

Listing 6.8: Using the Expect like interface of SSHShell

```
sh.send('echo hi')
r = sh.expect('hi')
# r == 0, the index of the matched pattern 'hi'

sh.send('echo hi')
r = sh.expect(['tie','hi','bye',sh.TIMEOUT])
# r == 1, the index of the matched pattern 'hi'

sh.send('echo hi')
r = sh.expect(['cry',sh.TIMEOUT])
# r == -1, no patterns matched, expect() timed out
```

The `expect()` method uses Python's regular expression module, `re`, and can accept complicated regular expressions as patterns. If the pattern argument contains a regular expression that is matched to data in the channel's buffer, the resulting `re.match` object is stored in the SSHShell object's `match` attribute, where the developer can query it further for the exact text and groupings that were matched. Listing 6.9 shows how to access matches found by the `expect()` method.

Listing 6.9: Using a regular expression as a pattern in the expect() method

```
# grab the prompt and escape it for use in a regular expression
import re
prompt = re.escape(sh.get_prompt())

# match any string
sh.send('echo hi')
sh.expect(['(.*){0}'.format(prompt)])
# the matching text is stored in a re.match object, available through sh.match
result = sh.match.groups()[0]
# result == 'hi\r\n'
```

6.3.2 Accessing a Hub's Tool Session Container Using SSH

Tool session containers on the hub can be accessed via an SSH connection, but SSH'ing into one can be a little tricky due to the hub configuration. All connections to the hub are routed through the the hub's web server, including connections destined for the web server and connections destined for tool session containers. For example, on the imaginary hub named `myhub.org`, a developer, with elevated permissions to login directly on the web server, would use the command shown in Listing 6.10 to login to the hub's web server.

Listing 6.10: SSH'ing into a hub's webserver

```
ssh user@myhub.org
```

In order to SSH into a tool session container hosted on the hub, the developer needs to use the hub's VirtualSSH proxy interface provided by the **session** command, as shown in Listing 6.11.

Listing 6.11: SSH'ing into a hub's tool session container

```
ssh -t -X user@myhub.org session
```

Normal users, without the elevated permissions needed to login on the web server, could use either command to connect to the hub, but in the case of Listing 6.10, the request would be automatically forwarded into a tool session container.

HUBcheck provides the `ToolSession` class to help developers connect to tool session containers through the hub's VirtualSSH proxy. The `ToolSession` class offers methods analogous to commands of the VirtualSSH proxy including the ability to start, stop, access, and list available tool session containers for a user with a password. Table 6.1 outlines the equivalent ToolSession methods for each of the Virtual SSH commands.

Starting a Tool Session Container With HUBcheck

VirtualSSH provides two ways to start a tool session container, using the **session create** or **session start** subcommands. The **session create** subcommand initiates the creation of a tool session container, allowing the developer to name the container

Table 6.1: HUBcheck's ToolSession class gives developers easy access to the hub's Virtual SSH Commands.

ssh [flags] [user@]hostname [command]	ts = ToolSession(host,port,username,password)
Virtual SSH Commands	ToolSession Object Methods
session create [session_title]	create(title=None)
session start	start()
session [session_number] [command]	access(snum=None,command=None)
session list	list()
session stop session_number	stop(session_number)
session help	help()

by setting the `session_title` parameter. After calling the **session create** subcommand, the user is returned to their local shell with a session number they can connect to. Similarly, the **session start** subcommand also initiates the creation of a tool session container, but goes the extra step of placing the user into the container where they can run shell commands on the remote host.

Listing 6.12: Starting a hub tool session container using VirtualSSH

```
ssh user@myhub.org session create
# 40023, a session number is returned to the user

ssh user@myhub.org session create mytitle
# 40023, a session number is returned to the user

ssh user@myhub.org session start
# user is placed into the tool session container
```

HUBcheck's `ToolSession` class provides access to these subcommands through the `create()` and `start()` methods. Just like VirtualSSH's **session create** subcommand, the `create()` method accepts an optional session title, but it returns three Paramiko `ChannelFile` objects that can be treated like Python file objects. One of the `ChannelFile` objects, `stdout`, holds the session number of the newly created session. The `ToolSession` class's `start()` method accepts no arguments and returns a `ToolSessionShell` object, which is derived from the `SSHShell` class introduced in Section 6.3.1. Listing 6.13 shows how to create and start tool session containers using HUBcheck's `ToolSession` class.

Listing 6.13: Starting a hub tool session container using the ToolSession class

```python
import hubcheck

ts = hubcheck.ToolSession(hostname,
                          username = username,
                          password = password)

(stdin,stdout,stderr) = ts.create()
# stdout.read() provides the session number

(stdin,stdout,stderr) = ts.create('mytitle')
# stdout.read() provides the session number

shell = ts.start()
# shell is a ToolSessionShell, a type of SSHShell
```

Accessing a Tool Session Container With HUBcheck

The default behavior of VirtualSSH's **session** command is to place the user into a tool session container. If the developer has multiple tool session containers running, they can choose which one to enter by providing the **session** command with an integer argument representing the session number, a unique integer identifier for a tool session container. The **session** command also accepts an optional `command` argument. When the `command` argument is provided, **session** will execute the command in the tool session container and return the user to the local shell along with the stdout and stderr streams from the command. Listing 6.14 shows how VirtualSSH's **session** command can be used to get into a tool session container with no arguments, with a session number, and with a command.

Listing 6.14: Accessing a hub tool session container using VirtualSSH

```
ssh user@myhub.org session
# user is placed into an open tool session container

ssh user@myhub.org session 40023
# user is placed into tool session container with session number 40023

ssh user@myhub.org session "echo hi"
# the command "echo hi" is run in a tool session container.
# "hi" is returned to stdout, user is returned to local shell

ssh user@myhub.org session 40023 "echo hi"
# the command "echo hi" is run in the tool
# session container with session number 40023
# "hi" is returned to stdout, user is returned to local shell
```

The `ToolSession` class provides tool session container access through the `access()` method. The `access()` method accepts two parameters, representing the session number and the command to run in the tool session container, just like VirtualSSH's **session** command. Example use of the `access()` method is shown in Listing 6.15.

Listing 6.15: Accessing a hub tool session container using the ToolSession class

```
shell = ts.access()
# places user into a tool session container and returns
# a ToolSessionShell object to control the container.

shell = ts.access(session_number=40023)
# places user into tool session container with
# session number 40023, and returns a ToolSessionShell
# object to control the container.

(stdin,stdout,stderr) = ts.access(command='echo hi')
# runs the command 'echo hi' in a tool session container
# returns stdin, stdout, and stderr to the user.

(stdin,stdout,stderr) = ts.access(40023,'echo hi')
# runs the command 'echo hi' in  tool session container
# with session number 40023. returns stdin, stdout, and
# stderr to the user.
```

Listing Available Tool Session Containers With HUBcheck

VirtualSSH's **session list** subcommand can be used to get a list of available tool session containers for a user. For each open tool session container, the command returns the session number, session name, and session title. The session name is the name of the tool that started the session. This is usually a workspace, but could be any of the installed tools on the hub since they all run in tool session containers. The **session list** subcommand also denotes the default tool session container for SSH connections by using a * in the output column named `Default`. Listing 6.16 shows example output from VirtualSSH's **session list** subcommand.

Listing 6.16: Listing available hub tool session containers using VirtualSSH

```
ssh user@myhub.org session list
#   Number  Default Name                 Title
#     8374     *    workspace_r1         Workspace (6:57 pm)
#     8584          workspace_r1         Workspace
# Connection to myhub.org closed.
# user is placed back in their local shell
```

This same information can be retrieved programmatically by using the `list()` method in the `ToolSession` class. Similar to running a command in a tool session container, the `list()` method returns its output as a 3-tuple whose elements represent the stdin, stdout, and stderr channels as Python file-like objects. Listing 6.17 shows an example of using the `ToolSession` class's `list()` method.

Listing 6.17: Listing available hub tool session containers using the ToolSession class

```
(stdin,stdout,stderr) = ts.list()
# returns the list of open sessions to the stdout variable

stdout.read()
#   Number  Default Name                 Title
#     8374     *    workspace_r1         Workspace (6:57 pm)
#     8584          workspace_r1         Workspace
```

To make accessing the information easier, the `ToolSession` class also provides the `get_open_session_detail()` method, which returns the same information as an iterable Python dictionary, with row numbers as keys and row data as values.

Listing 6.18: Iterating through tool session container details

```
import pprint

details = ts.get_open_session_detail()
# returns a dictionary of open session data

pprint.pprint(details)
#{0: {'default': True,
#     'name': 'workspace_r1',
#     'session_number': '8374',
#     'title': 'Workspace (6:57 pm)'},
# 1: {'default': False,
#     'name': 'workspace_r1',
#     'session_number': '8584',
#     'title': 'Workspace'}}

for row in details.values():
  if row['session_number'] == '8584':
    title = row['title']

print title
# Workspace
```

Stopping Tool Session Containers With HUBcheck

VirtualSSH allows users to stop a tool session container using the **session stop** subcommand. The command accepts an integer argument that specifies the session number that should be stopped.

Listing 6.19: Stopping a hub tool session container using VirtualSSH

```
ssh user@myhub.org session stop 40023
# stopping session 40023
# Connection to myhub.org closed.
# user is placed back in their local shell
```

To stop tool session containers using the `ToolSession` class, use the `stop()` method. The `stop()` method accepts a single parameter, an integer session number specifying the tool session container to stop.

Listing 6.20: Stopping a hub tool session containers using the ToolSession class

```
(stdin,stdout,stderr) = ts.stop(40023)
```

6.3.3 Managing Tool Session Containers

When writing automated scripts and tests, keeping track of all of the tool session containers being opened and closed can be a hassle. HUBcheck tries address this by offering the `ContainerManager` class, which promotes the efficient reuse of tool session containers when possible. The ContainerManager class is a singleton that can be used to create, access, and stop tool session containers for multiple users.

Consider the case where several test cases need access to a tool session container to perform a test. The simple solution would be to have each test case start, access, and stop a new tool session container to perform its test. This approach provides isolation between each test, helping ensure another resource doesn't accidentally close the tool session container while a test case is using it. It is, however, terribly inefficient. Starting up a tool session container takes a few seconds and using it for a single, non-destructive test would be wasteful. Often, an execution of HUBcheck runs hundreds of tests, the majority of which only query resources in the tool session container, leaving it in good condition for further use. The approach taken by many HUBcheck based tools is to reuse tool session containers whenever possible.

The `ContainerManager` class helps implement a tool session container reuse approach. Accessing tool session containers is similar to using the `ToolSession` class directly, but removes most of the rarely used features. The `ContainerManager` class provides an `access()` method developers can use to enter a tool session container. The `access()` method takes three parameters, the hostname of the hub hosting the tool session container, the username and the password of the user opening the tool session container. Given this information, the `ContainerManager` class looks in its internal dictionary to see if it already has a tool session container open for the hostname and username combination. If it does, a new shell for that container is opened and returned to the user as a `ToolSessionShell` object. If not, a new tool session container is created and a `ToolSessionShell` object is returned.

Listing 6.21: Accessing a tool session container using the ContainerManager class

```
import hubcheck

cm = hubcheck.ContainerManager()

ws1 = cm.access(hostname,username,password)
# ws1 is a ToolSessionShell object

session_number1 = ws1.execute('echo $SESSION')
# '40023'
```

The `ContainerManager` can track multiple open tool session containers, by multiple users, on multiple hubs. Calling the `access()` method a second, or a third, time with the same hostname and username results in additional shells being opened in the same tool session container.

Listing 6.22: Multiple accesses to a tool session container using the ContainerManager class

```
ws2 = cm.access(hostname,username,password)
# ws2 is another ToolSessionShell object

session_number2 = ws2.execute('echo $SESSION')
# '40023'
```

Calling the `access()` method with a different hostname or username results in a new tool session container being created.

Listing 6.23: ContainerManager can handle multiple users' tool session containers

```
ws3 = cm.access(hostname,username2,password2)
# ws3 is another ToolSessionShell object

session_number3 = ws3.execute('echo $SESSION')
# '40024'
```

In many HUBcheck based tools, there is little advantage to closing a tool session container before the program ends, but for the times when this is needed, the `ContainerManager` class provides the `stop()` and `stop_all()` methods. The `stop()` method uses its parameters, a hostname, username, and session number, to determine which tool session container to stop. The `stop_all()` method loops through all tool session containers managed by the `ContainerMananger` object and stops them.

Listing 6.24: Stopping a tool session container using the ContainerManager class

```
cm.stop(hostname,username,session_number)
# stop username's tool session container on host hostname
# with session number session_number

cm.stop_all()
# stop all tool session containers managed by the cm object.
```

6.3.4 Interacting With the Tool Session Container

A `ToolSessionShell` object is returned for many of the methods that access a tool session container. The `ToolSessionShell` class is derived from the `SSHShell` class and provides the `send()` and `expect()` methods described in Section 6.3.1. It also provides the `SSHShell`'s `execute()` method, which combines both the `send()` and `expect()` methods into one function call that checks the exit status of the executed command. If a command returns a non-zero exit status, an `ExitCodeError` exception is raised, and command execution stops. In this respect, the `execute()` method acts like a shell with the **-e** flag set, where a script will exit immediately upon error. A successful call to the `execute()` method returns the output of the command and the exit status of executing the command.

Listing 6.25: Interacting with the tool session container using the ToolSessionShell class

```
import hubcheck

cm = hubcheck.ContainerManager()

ws = cm.access(hostname,username,password)
# ws is a ToolSessionShell object

out,es = ws.execute('echo hi')

print out
# 'hi'
```

The `ToolSessionShell` class provides features specific to working in the shell of a tool session container. It provides the `importfile()` method to transfer files from the user's desktop into the tool session container, the `exportfile()` method to transfer files from the tool session container to the user's desktop, and a few functions to simplify parsing tool session container resource files.

6.3.5 Transferring Files Between The User Desktop and the Hub

The hub supports three methods of transporting files between the user desktop and the user's hub account, including `webDAV`, `filexfer`, and `sftp`. HUBcheck provides ways to use `filexfer` and `sftp`, while the Python module `webdavlib` [37] provides a reliable client-side interface for the `webDAV` [38] protocol.

The hub's `filexfer` protocol consists of two commands, **importfile** and **exportfile**. Filexfer transfers start on the command line of a tool session container's X terminal, by issuing the the **importfile** command to transfer files from the user's desktop into their hub account, or the **exportfile** command to transfer files from the user's hub account into their desktop. The `filexfer` commands use the tool session container's **clientaction** program to initiate a popup window in the user's web browser. When importing a file, the user populates the popup window with the text or filename they would like transferred into their hub account. After submitting the form in the popup window, the data is then saved in the hub account. When exporting a file, the popup window contains the data of the file from the user's hub account.

To implement simple file transfers, the `ToolSessionShell` class provides the `importfile()` and `exportfile()` methods. Both methods take two arguments representing a local filename from inside of the workspace and a remote filename from the user's desktop. The direction of transfer is determined by the method being called. Under the hood, these two functions approximate the actions being performed by the hub's `filexfer` commands, by using the `sftp` protocol to transfer the files. It is possible to use HUBcheck's web module in coordination with the `ToolSessionShell` class to more closely imitate what users would really do on the hub, but the goal of the implementation is to transfer files and, on the command line, using `sftp` is more efficient.

Listing 6.26: Transferring files using ToolSessionShell's importfile and exportfile methods

```
import hubcheck

cm = hubcheck.ContainerManager()

ws = cm.access(hostname,username,password)
# ws is a ToolSessionShell object

# importing a file from the desktop to the hub account
ws.importfile('desktop_file.txt', 'hub_file.txt')

# importing data from the desktop to a file in the hub account
file_data = 'transferring is easy'
fsize = ws.importfile(file_data, 'hub_file.txt', is_data=True)
assert fsize == len(file_data)

# exporting a file from the hub account to the desktop
ws.exportfile('hub_file.txt', 'desktop_file.txt')
```

For more control over how files are transferred, the HUBcheck library also provides access to Paramiko's `SFTPClient` class. HUBcheck's `SFTPClient` class is a small wrapper class around the Paramiko `SFTPClient` class, which can be used to access a user's hub account through the `sftp` protocol. With this method, users have full access to `sftp` functions like `get`, `put`, `remove`, `chmod`, `open`, `chdir`, and more.

Listing 6.27: Transferring files using the SFTPClient class

```
import hubcheck

sftp = hubcheck.SFTPClient(hostname,username=username,password=password)
# sftp is a Paramiko SFTPClient object

# transfer file from the hub account to the desktop
sftp.get('hub_file.txt','desktop_file.txt')

# transfer file from the desktop to the hub account
sftp.put('desktop_file.txt','hub_file.txt')

# close the sftp connection
sftp.close()
```

6.4 Building Applications Backed by the HUBcheck Library

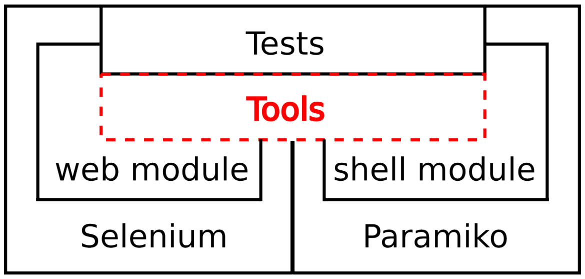


Fig. 6.4.: Command line utilities can be built on top of the HUBcheck library by using the `hubcheck.Tool` class.

The HUBcheck library includes several programs that build upon the web and shell modules. While each program has a different goal, there are features common to all of them, like configuration options and environment setup, that are tedious to write for one program and inefficient to copy for multiple programs. For this reason, HUBcheck includes the `hubcheck.Tool` class, which can be subclassed to get a common set of command line and configuration file options with an intuitive parser, automatic logging setup, a virtual display for running web browser based automation, and a web proxy to help monitor, block and analyze communications between a web browser and a web site.

Below, we explore the features of `hubcheck.Tool` based programs by building an example tool that performs a user login through the hub's web and SSH interfaces.

6.4.1 Building a Tool

`hubcheck.Tool` is a base class for command line tools that use the HUBcheck library. The class provides many of the boilerplate features that are repeated in HUBcheck based programs. By subclassing `hubcheck.Tool`, users can quickly create a feature rich command line tool for automating web and shell access on the hub.

Listing 6.28: HUBcheck based tools follow this general template

```
import hubcheck

class LoginTool(hubcheck.Tool):
    def __init__(self,logfile='hcutils.log',loglevel='INFO'):
        super(LoginTool,self).__init__(logfile,loglevel)
        # introduce new command line and configuration options

        # parse command line / config file options, start logging
        self.parse_options()
        self.start_logging()

    def command(self):
        # add code that does the work

if __name__=='__main__':
    tool = LoginTool()
    tool.run()
```

There are three steps to creating a new HUBcheck based tool:

1. Subclass `hubcheck.Tool` to create a new tool class.

2. Populate the tool's `__init__()` and `command()` methods.

3. Call the `run()` method of an instance of the tool's class.

Listing 6.28 shows the general template for HUBcheck based tools. The template starts by subclassing `hubcheck.Tool` into a class named `LoginTool`, the name of the tool we are creating. `hubcheck.Tool`'s `__init__()` method takes care of setting up command line and configuration file option parsing, so it is important that it is called early in the object creation process. In Listing 6.28, this happens in line 5. `hubcheck.Tool`'s `__init__()` method sets up five variables that can be used by the `LoginTool` class:

1. `self.command_parser`
2. `self.config_parser`
3. `self.options`
4. `self.logger`
5. `self.testdata`

The first two, `self.command_parser` and `self.config_parser`, are parsers for options set on the command line or in an INI-style configuration file. You can manipulate these variables inside of `__init__()`, using their `add_argument()` and `add_option()` methods respectively, to add new options that the parsers will recognize.

Listing 6.29: Use the `add_argument()` method to define new command line flags for the tool

```
1   def __init__(self,logfile='hcutils.log',loglevel='INFO'):
2       ...
3       # introduce new command line and configuration options
4       self.command_parser.add_argument(
5           '--video-filename',
6           help='name of the video file',
7           action="store",
8           dest="videofn",
9           default='password_change.mp4',
10          type=str)
11      ...
```

The `hubcheck.Tool` class includes functionality to record the virtual display where the web browser window is running, but does not expose the ability to name the file that the recording is saved to. Adding a `--video-filename` flag to a tool allows the user to specify the name of the file where video recordings should be saved. Listing 6.29 shows an example of adding the new `--video-filename` flag to the command line parser. With the new flag in place, the program can be invoked as shown in Listing 6.30.

Listing 6.30: Invoking a HUBcheck based tool with a new command line option

```
./logintool --config hub.conf --video-filename myvideo.mp4 testuser2
```

The `self.parse_options()` method is called to perform the command line and configuration file option parsing. It resolves conflicts and stores the collected options in the variable named `self.options`. In Listing 6.29, line 8 defines the value obtained from the `--video-filename` command line argument to be stored in the variable `self.options.videofn`. Before exiting initialization, `__init__()` starts a logger, another benefit of subclassing the hubcheck.Tool class. From within the tool, the logger is accessible through the `self.logger` variable.

The `command()` method is responsible for performing the main tasks of the program. The method is called indirectly when the object's `run()` method is executed. In the template shown in Listing 6.28, this is done at the end of the script on line 17. The run() method takes care of much of the setup and teardown of the environment from which the automation takes place. It is responsible for loading the HUBcheck configuration data, setting up directories for browser screenshots and videos, starting a virtual display for the browser to run in, and launching a web proxy for the web browser. The `run()` method prepares the environment so the tasks in the `command()` method can launch a browser with minimal additional system configuration. After setting up the environment, `run()` calls the tool's `command()` method.

A HUBcheck tool's `command()` method holds the objectives of the tool. Generally, the `command()` method starts by evaluating the command line and configuration file options, setting local variables based on the parsed options.

Listing 6.31: The command() method of a HUBcheck based tool

```
1     def command(self):
2         # set variables based on parsed options
3         username = self.options.remainder[0]
4         videofn = self.options.videofn
5
6         # retrieve account information
7         userpass = self.testdata.find_account_password(username)
8
9         # grab hub configuration from the testdata file
10        locators    = self.testdata.get_locators()
11        hostname    = self.testdata.find_url_for('https')
12        url = "https://%s" % (hostname)
13
14        # create a hubcheck object
15        hc = hubcheck.Hubcheck(hostname=hostname,locators=locators)
16
17        # initialize recording
18        self.start_recording_xvfb(videofn)
19        ...
```

In the `command()` method for our example **logintool** program, shown in Listing 6.31, lines 9 - 11 introduce the use of the `self.testdata` variable, which is setup by the `hubcheck.Tool` class. `self.testdata` is an instance of HUBcheck's `TestData` class, which provides helper methods for querying information from a HUBcheck configuration file. `self.testdata` gives developers access to information about which HUBcheck web element locators, test user account information, and hub URLs to use. With hub specific information acquired, the program creates a HUBcheck browser object in line 15 and starts recording the virtual display in line 18.

The next step is to login to the hub through the web interface. Listing 6.32 uses the `hc` object's `browser` and `utils` attributes to open a web browser, login to the web site, logout of the web site, and close the browser.

Listing 6.32: Login through the web interface

```
def command(self):
    ...
    # start up a selenium webdriver based browser
    hc.browser.get(url)

    # login to the hub using the web interface
    hc.utils.account.login_as(username,userpass)

    # navigate to the dashboard and logout
    hc.utils.account.logout()

    # close the browser and cleanup
    hc.browser.close()
    self.stop_recording_xvfb()
    ...
```

Similarly, Listing 6.33 uses a `ToolSession` object to login to a tool session container through the SSH interface.

Listing 6.33: Login through the Virtual SSH interface

```
    def command(self):
        ...
        # login to the hub using the Virtual SSH interface
        ts = hubcheck.ToolSession(
              hc.hostname, username=username, password=userpass)

        # SSH into a tool container and run the 'echo hi' command
        stdin,stdout,stderr = ts.access(command='echo hi')

        # check stdout for the output of the command, 'hi'
        output = stdout.read(1024)
        assert output == 'hi\n', \
            "error ssh'ing into tool container: %s" % (output)
```

After the `command()` method exits, control is returned to the `run()` method, which stops the web proxy, shuts down the virtual display, and performs cleanup actions in the environment.

6.4.2 Example Tools

The `command()` method for the **logintool** program is pretty elementary, but any task can be substituted in. Nearly all of the tools in the HUBcheck library use this subclassing and `command()` method design pattern as a foundation for the tool's operation. Listed below are a few tools built upon the HUBcheck library.

Nightly Rappture Builds

The Rappture Toolkit [39] is a library that helps people build and deploy simulation tools with graphical user interfaces. The library includes a set of Tcl/Tk based graphical user interface widgets and language bindings for communicating with the GUI in C/C++, Fortran, Ruby, Matlab/Octave, Java, Perl, and Python. One part of Rappture testing includes building and exercising the library inside of a hub tool session container. To perform these actions, the HUBcheck library is used to get into a hub's tool session container, build the Rappture Toolkit, and run a number of test suites on the build. This is the job of the HUBcheck Nightly Rappture Build script, **hcnrb**. Once completed, the nightly builds are transferred to the `rappture.org` website where users can download precompiled or source versions of the library on a nightly basis.

Test User Tools

HUBcheck relies on a number of test user accounts setup with different configurations. Currently, these test accounts are managed in the same way regular user accounts are managed, through the hub's website interface. To help manage these accounts' properties, a number of HUBcheck based programs have been written including an account registration tool, password updating tool, and a profile management tool.

register_account is a program built to fill out the account registration web page. It is generally used shortly after a new hub installation, to register HUBcheck's test user accounts. Account registration is a two step process and the **register_account** program can be applied to the first step, filling out the new account registration form. Most hub registration forms incorporate a CAPTCHA to keep robots from registering accounts. **register_account** does not attempt to interpret the CAPTCHA. Instead, it leaves time for a human to solve the CAPTCHA. After filling out the new account registration form, the hub sends a confirmation email to the user, and the user responds by clicking the link in the email. This feature is not available in the current version of the **register_account** program, but could show up in a future version. The **register_account** script uses the HUBcheck library's page objects to manage web page navigation and to populate the account registration web form.

After test accounts have been created, the focus shifts to maintaining the accounts. Account maintenance is important in helping reduce the number of false positives when running tests and to ensure programs can gain the access they need to accomplish their automated tasks. One of the essential tasks for managing test accounts is to keep them secure, which includes frequently changing their passwords. The hub provides a web form for users to change their passwords and HUBcheck has a page object to automate its use. The **hcpwc** program uses the HUBcheck library's page objects to help automate the generation and updating of test account passwords on the hub.

Managing the user profile is an account maintenance task that needs to be performed at least once in the life of the account. The hub user profile holds information that is usually collected for the purposes of identifying types of users to the hub's funding agencies, e.g. the National Science Foundation. Sometimes this information is collected during the account registration process. Including the information on the new account registration form could deter people from signing up. As a result the information could be requested later, when a user wants to use what may be considered a premium service on the hub. When testing hub components, having account profile update requests popup unpredictably on a web page may contribute to having false positives in test results. A solution to this problem is to fully populate the user profile for all test accounts just after registering the accounts. This is what the **hcuserprofile** program does. The components of the hub user profile are a configurable, yet finite set. **hcuserprofile** uses HUBcheck's page objects to navigate to the user profile web page and query the page for a hard coded list of components that are typically included in hub user profiles. When a component is found, **hcuserprofile** attempts to provide a reasonable response for the component.

Test Runner

One of the original motivations for writing HUBcheck was to test the hub. While HUBcheck has become more than just testing, the test runner is still one of the most heavily used tools in the library. HUBcheck's test runner tool, **hctestrunner**, is a wrapper around **pytest**, a mature full-featured Python testing library. **hctestrunner** uses the `hubcheck.Tool` class to manage the environment in which test cases are run. The program can be executed from the command line and requires a HUBcheck configuration file to work.

Listing 6.34: hctestrunner accepts hubcheck.Tool flags and pytest flags

```
hctestrunner --config ./hubzero.conf -m nightly --collect-only
```

The **hctestrunner** program accepts the normal set of command line flags inherited from the `hubcheck.Tool` class, and also accepts command line flags for **pytest**. Listing 6.34 shows how to retrieve a list of tests that are marked with the tag "nightly" and would run on `hubzero.org`. To accomplish this, we provide the hubzero.conf configuration file through `hubcheck.Tool`'s `--config` flag, specify the nightly mark through **pytest**'s `-m` flag, and ask **pytest** to only collect tests (not run them) with the `--collect-only` flag. **hctestrunner** parses all of the flags, but passes any flags it does not recognize on to **pytest**. After setting up the web proxy, virtual display and evaluating command line and configuration options, **hctestrunner** calls on **pytest** to manage searching for and running test cases.

6.5 Writing Tests Using the HUBcheck Library

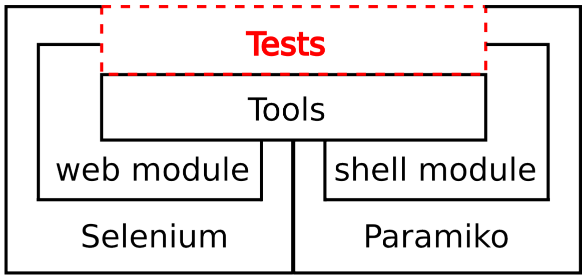


Fig. 6.5.: One of HUBcheck's most used features is its test runner and its ability to be embedded within tests.

HUBcheck provides the **hctestrunner** tool to manage the collection and execution of test cases. Under the hood, **hctestrunner** depends on **pytest**, a flexible Python based test case management library that can handle test cases written in a number of formats including `unittest` [40], `nose` [41], and `doctest` [42]. Section 6.4 explored writing programs using the HUBcheck library, and while test cases can also be written in the same way, this approach can lead to an excess of code and resource repetition between test cases. In the following sections, we describe ways to write robust, easy to understand test cases using the HUBcheck library.

6.5.1 Test Fixtures

One of the strengths of the **pytest** library is its flexibility. Many test case management libraries include a feature often referred to as a *test fixture* [43]. Test fixtures are functions, or bits of code, that place the software under test in a fixed state as a baseline for running tests.

Traditionally, x-unit [44] based testing libraries have provided two types of fixtures: setup fixtures run before a test case, and teardown fixtures run after the test case. **pytest** provides a number of different fixture options which can call upon each other or be reused within the class, module, or project scopes.

6.5.2 The TestCase2 Class

HUBcheck provides the `TestCase2` class whose purpose is similar to that of the `Tool` class mentioned in Section 6.4.1. The `TestCase2` class provides much of the environment setup and teardown code needed to run a test and document its behavior in a more detailed manner than just pass or fail. The `TestCase2` class is responsible for setting up the filename for browser screenshots, recording individual test cases to separate video files, setting up test case testdata in local variables, and managing the page object catalog. The class also initializes a local web browser object for each test case. The `TestCase2` class provides each test case with the following local variables:

1. `self.browser`

2. `self.catalog`

3. `self.utils`

4. `self.testdata`

5. `self.locators`

6. `self.https_uri`

7. `self.https_authority`

8. `self.http_authority`

`TestCase2` hooks into `pytest`'s `setup_method()` and `teardown_method()` test fixtures to provide features similar to that of HUBcheck based tools.

6.5.3 Building a Test Case

Consider rewriting the hub website login example, from Listing 6.6, as a test case. Listing 6.35 demonstrates how to setup a `TestCase2` based test case. Just like the `Tool` class, to use the `TestCase2` class the user first subclasses it. All classes derived from `TestCase2` have `setup_method()` and `teardown_method()` fixtures, even if it is not explicitly stated in the derived class as is the case for the `TestHubLogin` class from Listing 6.35. These two fixtures allow `TestCase2` to perform its variable, browser, page object catalog, and utility setup before the test case is run, and teardown after the test case has completed. In accordance with `pytest`'s test case naming conventions, all methods of the derived class with names prefixed by *test_* are considered test cases. In Listing 6.35, this includes the `test_website_login()` method. The test case performs four tasks. First, it grabs a username and password from the testdata file in line 4. Next, it navigates the web browser to the hub's homepage in line 8. In Line 10, it submits a populated hub login form. Lastly, in line 14, it checks that the login was successful.

Listing 6.35: Hub website login using HUBcheck's TestCase2 class

```
class TestHubLogin(hubcheck.TestCase2):
    def test_website_login(self):
        """ try to login to the hub website """
        self.username,self.userpass = \
            self.testdata.find_account_for('registeredworkspace')

        # setup a web browser
        self.browser.get(self.https_authority)

        self.utils.account.login_as(self.username,self.userpass)

        # verify you have successfully logged in
        po = self.catalog.load_pageobject('GenericPage')
        assert po.header.is_logged_in(),'Login Failed'
```

It is interesting to note how clean and easy to read the `test_website_login()` test case is when compared to the login script in Listing 6.1 or even the original login script in Listing 5.7. Part of the test case's simplicity is due to its use of the `TestCase2` class, which wraps up most of the standard code needed to setup and teardown the web browser, browser recording, testdata, page object catalog, and utilities. Test cases that are easy to read are often easy to understand and maintain. The `TestCase2` class helps keep test cases short and to the point.

Adding more test cases to the `TestHubLogin` class is also easy and helps demonstrate the power of `pytest`'s fixtures. A complementary test to logging into the website is logging out. As you might have suspected, there is a bit of code overlap between the login and logout tests. For example, in order to test login and logout, both tests need to login. This common code can be placed in the `setup_method()` fixture. In Listing 6.36, the code to retrieve the test user credentials, navigate the browser to the hub's homepage, and login has been abstracted out of the `test_website_login()` test case, and placed in the `TestHubLogin` class's `setup_method()` fixture. This fixture and `TestCase2`'s `setup_method()` fixture don't collide as they would in a normal inheritance situation. Instead, through the magic of metaclasses, `TestCase2` recognizes the existence of `TestHubLogin`'s `setup_method()` method, and embeds the method inside of its `setup_method()` fixture. By the time `TestHubLogin`'s `setup_method()` is called, `TestCase2` has already completed setting up the environment and variables for the test case. This same embedding happens with `TestHubLogin`'s `teardown_method()`, but in reverse. `TestCase2` recognizes `TestHubLogin`'s `teardown_method()` method, calls the method, and then proceeds to calling its own `teardown_method()` fixture.

Listing 6.36: Using pytest fixtures while subclassing the TestCase2 class

```
class TestHubLogin(hubcheck.TestCase2):
    def setup_method(self,method):
        self.username,self.userpass = \
            self.testdata.find_account_for('registeredworkspace')

        # setup a web browser
        self.browser.get(self.https_authority)

        # login to the hub website
        self.utils.account.login_as(self.username,self.userpass)
```

With the `setup_method()` fixture in place, the new `test_website_login()` test case is even more compact and to the point. After the `setup_method()` fixture runs and logs the user into the hub website, the `test_website_login()` test case just checks to see if the user has successfully logged in. Listing 6.37 shows the new `test_website_login()` test case.

Listing 6.37: New website login test case, using the setup_method() fixture

```
class TestHubLogin(hubcheck.TestCase2):
    ...
    def test_website_login(self):
        """ try to login to the hub website """
        # verify you have successfully logged in
        po = self.catalog.load_pageobject('GenericPage')
        assert po.header.is_logged_in(),'Login Failed'
```

Similarly, the new `test_website_logout()` test case, in Listing 6.38, verifies that the login was successful, then attempts to logout of the website. Finally, it checks that logging out of the website was successful.

Listing 6.38: New website logout test case, using the setup_method() fixture

```
class TestHubLogin(hubcheck.TestCase2):
    ...
    def test_website_logout(self):
        """ try to logout of the hub website """
        # verify you have successfully logged in
        po = self.catalog.load_pageobject('GenericPage')
        assert po.header.is_logged_in(),'Login Failed'

        # logout of the website
        po.header.goto_logout()
        assert not po.header.is_logged_in(),'Logout Failed'
```

Writing test cases using the HUBcheck library is an extension of writing programs that use the library. Many features are analogous, including setting up the environment, starting the web browser, starting a web proxy, setting up a recordable virtual framebuffer, and loading testdata. Much of the boilerplate code for setup and teardown is taken care of by the **hctestrunner** tool and the `TestCase2` class. When used together, developers are able to quickly write easy to read, robust test cases.

6.5.4 HUBcheck's Test Suites

Test cases can be grouped into test suites by using a feature from **pytest** called *marks*. **pytest** marks are a way of tagging or marking tests. **pytest** provides an easy way to collect all tests with a specific mark or a logical combination of marks using the `-m` flag. Tests that come with the HUBcheck library are tagged with at least one of several popular marks that denote when the test should be run including *nightly*, *weekly*, *upgrade*, *prod_safe_upgrade*, *reboot*, and *hcunit*.

7. BUILDING PAGE OBJECTS FOR HUBCHECK

The Page Object design pattern is the cornerstone in creating compact easy to read automation scripts, and it is key in combating brittle test cases. The pattern uses object methods to represent the services offered on a web page. By doing this, the pattern encourages automation developers to move the low level details of how a task is completed out of automation scripts and into libraries. This leaves the automation script with abstract generalizations of what should happen and calls to the library that make those generalizations happen.

In this chapter, we briefly review why the Page Object design pattern is important, dive into how to use HUBcheck's page objects, and learn how to build new page objects that work with the HUBcheck library. We'll also investigate three new design patterns that are helpful when building page objects for HUBcheck.

7.1 Review of the Page Object Design Pattern

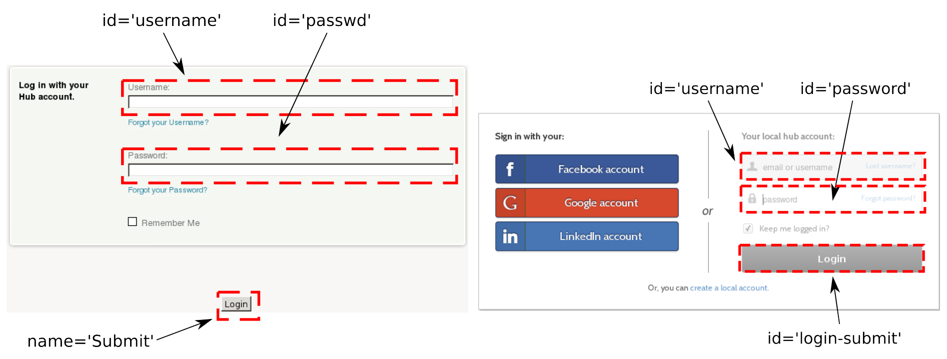
Listing 7.1: Login automation script for hubzero.org

```
from selenium import webdriver

# start the browser and navigate to the login page
browser = webdriver.Firefox()
browser.get('https://hubzero.org/login')

# perform the login action
browser.find_element_by_css_selector("#username").clear()
browser.find_element_by_css_selector("#username").send_keys("testuser")
browser.find_element_by_css_selector("#passwd").clear()
browser.find_element_by_css_selector("#passwd").send_keys("abc123")
browser.find_element_by_css_selector("[name='Submit']").click()
```

In Section 5.1.6, we first introduced an automation script that performed a user login on a hub, noting a few undesirable patterns within the script. First we identified repetition in searching for the username and password fields. Second we noted the pattern of clearing the field before sending data to it. While it isn't always necessary to clear an input before sending data to it, this tends to be good practice that helps remove previously set values from fields. One pattern we didn't note has to do with the login action on a larger scale. The action as a whole takes about five lines of code to populate the username field, populate the password field, and click the login button. The problem arises when the login service is used in multiple scripts. Writing these five lines of code over and over is costly and is prone to variation. Furthermore, it adds to the brittleness of all of the automation scripts.

Consider the case where the developer would like to use a set of automation scripts, with these five lines of login code, on another hub website, like nanohub.org. The developer may find that while the scripts worked on hubzero.org, things are a little different on nanohub.org. In particular, the element locators for the password input field and login button on hubzero.org don't match the ones on nanohub.org. On hubzero.org, the password input field is identified by the css selector `#passwd`, while on nanohub.org, the same field is identified by the css selector `#password`. Similarly, on hubzero the login button is identified by the css selector `[name='Submit']`, while on nanohub.org it is identified by the css selector `#login-submit`.

With every new hub the automation script is taken to, there is the opportunity for either of the previous element locator sets to be used, or even a new set. Keeping separate automation scripts for each hub is a good way to encourage divergence of the source. Instead the script should be written in a way where the differences are abstracted away. This is one of the goals of the Page Object design pattern.



(a) hubzero.org login form. (b) nanohub.org login form.

Fig. 7.1.: The variation in locators used on the hub login page causes an additional level of complexity when trying to develop automation scripts generic enough to work across the hubs managed by the HUBzero team.

Listing 7.2: Page object for the hub /login page

```
# pageobjects.py
locdict = {
  'hubzero' : {
    'username' : '#username',
    'password' : '#passwd',
    'login_b'  : "[name='Submit']"
  },
  'nanohub' : {
    'username' : '#username',
    'password' : '#password',
    'login_b'  : '#login-submit'
  }
}

class LoginPage(object):
    def __init__(self,browser,hub):
        self.username = \
            browser.find_element_by_css_selector(locdict[hub]['username'])
        self.password = \
            browser.find_element_by_css_selector(locdict[hub]['password'])
        self.login_b = \
            browser.find_element_by_css_selector(locdict[hub]['login_b'])

    def login_as(self,username,password):
        self.username.clear().send_keys(username)
        self.password.clear().send_keys(password)
        self.login_b.click()
```

Listing 7.2 shows an example page object for the hub login page that can be configured to work with hubzero.org or nanohub.org. The `locdict` variable holds dictionaries of locators that are used by the `LoginPage` class. The user specifies which hub locators they want to use in the `LoginPage` constructor.

Listing 7.3: Login automation script for hubzero.org, using page objects

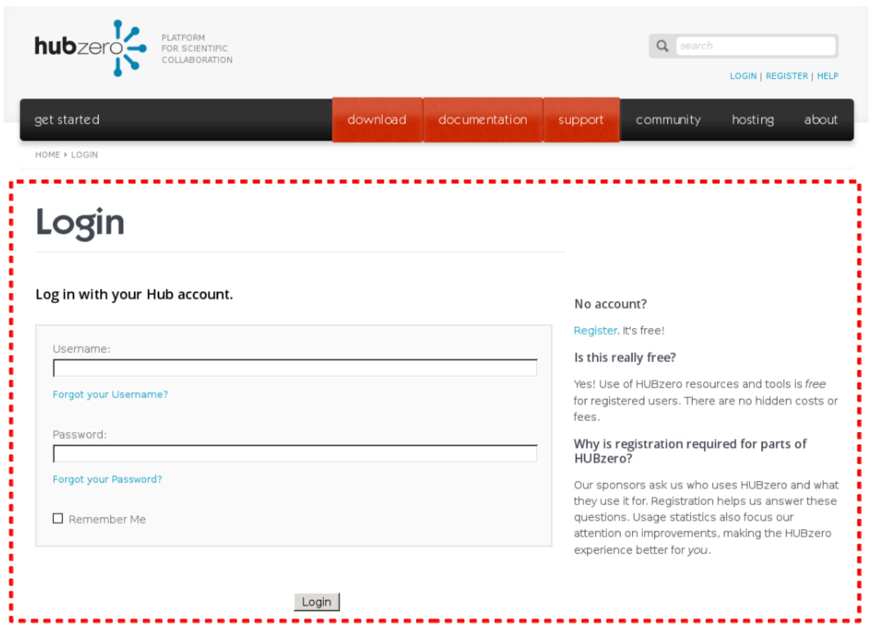
```
1 from selenium import webdriver
2 from pageobjects import LoginPage
3
4 # start the browser and navigate to the login page
5 browser = webdriver.Firefox()
6 browser.get('https://hubzero.org/login')
7
8 # perform the login action
9 po = LoginPage(browser,hub='hubzero')
10 po.login_as(username="testuser",password="abc123")
```

In line 9 of Listing 7.3, the page object is created and configured to use the `hubzero` locators. Next, the automation script calls the page object's `login_as()` method to perform the login service on the page.

The page object solution is flexible and robust, allowing the developer to update the page object to handle more hubs by adding locators to the `locdict` variable. The new automation script can easily be configured to work on nanohub.org, or any other hub recognized by the page object. Additionally, all of the code to perform the login action is in one place, the `login_as()` method. If there is a change to how the login service works, an update to the `login_as()` method updates all of the scripts that use the method. The HUBcheck library provides classes for popular HTML elements to help developers quickly build page objects. Let's explore how to rebuild the LoginPage page object by using the HUBcheck library.

7.2 Rebuilding the Login Page Object With HUBcheck

Before representing a web page as a page object, the developer must understand the purpose of the web page. Web pages provide services. When a user navigates to a web page, they are either receiving information from the system or providing information to the system. On the hub, the login web page provides four services. The most recognized service is providing user authentication for accessing personalized and restricted material on the hub. The other services provided by the page are to guide users to the registration page, the username reminder page, and the password reset page.



Log in with your Hub account.

Forgot your Username?

Forgot your Password?

No account?

Register. It's free!

Is this really free?

Yes! Use of HUBzero resources and tools is *free* for registered users. There are no hidden costs or fees.

Why is registration required for parts of HUBzero?

Our sponsors ask us who uses HUBzero and what they use it for. Registration helps us answer these questions. Usage statistics also focus our attention on improvements, making the HUBzero experience better for *you*.

Fig. 7.2.: The part of the web page represented by the Login widget is outlined in red.

The login web page can be split into three sections. The header section is the top portion of the page available on all hub web pages. This includes site navigation links and banners. The footer section is the bottom portion of the page, also available on all hub web pages. This section includes copyright, contact, and site ownership information. The rest of the page can be considered the Login widget, and is shown in

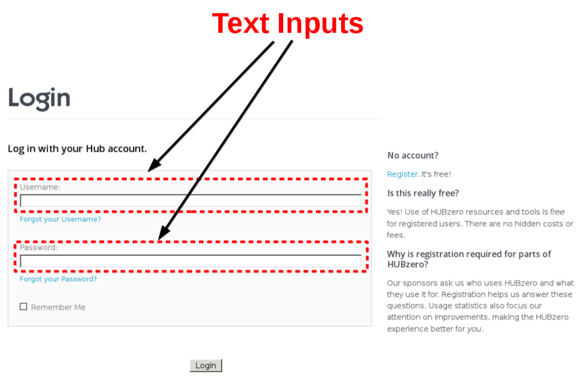
Figure 7.2. The Login widget is a mega widget, a widget made up of smaller widgets or elements, like links, text boxes, checkboxes, and buttons. Each of these elements are widgets in their own right, and the classes that represent them provide a few specialized services.

7.2.1 Matching Web Page Widgets to HUBcheck Page Object Classes

On the login page, each element can be represented by a HUBcheck page object class. Links are represented by the `Link` class, input text boxes are represented by the `Text` class, checkboxes are represented by the `Checkbox` class, and buttons are represented by the `Button` class. Listing 7.4 starts to build a new `Login` page object by identifying the widgets that are available on the web page and matching them up with comparable HUBcheck classes. In the code, the username and password text boxes are represented by `Text` objects. The username reminder, password reset, and register links are represented by `Link` objects. The remember me checkbox is represented by a `Checkbox` object and the form submission button is represented by a `Button` object. HUBcheck includes nine classes that represent popular HTML elements including `Button`, `Checkbox`, `Link`, `Radio`, `Select`, `Text`, `TextReadOnly`, `TextAC`, `TextArea`, and `Upload`.

Listing 7.4: The Login class is a composition of other classes representing the elements on the web page.

```
class Login(BasePageWidget):
    def __init__(self):
        ...
        self.username  = Text(self,{'base':'username'})
        self.password  = Text(self,{'base':'password'})
        self.remember  = Checkbox(self,{'base':'remember'})
        self.remind    = Link(self,{'base':'remind'})
        self.reset     = Link(self,{'base':'reset'})
        self.register  = Link(self,{'base':'register'})
        self.submit    = Button(self,{'base':'submit'})
        ...
```



Login

Log in with your Hub account.

Username:

Forgot your Username?

Password:

Forgot your Password?

Remember Me

Login

No account?

Register. It's free!

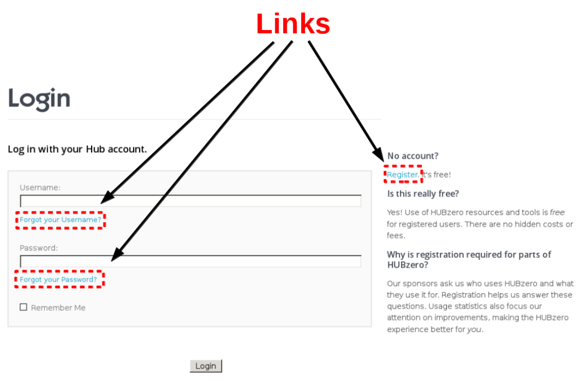
Is this really free?

Yes! Use of HUBzero resources and tools is *free* for registered users. There are no hidden costs or fees.

Why is registration required for parts of HUBzero?

Our sponsors ask us who uses HUBzero and what they use it for. Registration helps us answer these questions. Usage statistics also focus our attention on improvements, making the HUBzero experience better for *you*.

(a) Text box widgets



Login

Log in with your Hub account.

Username:

Forgot your Username?

Password:

Forgot your Password?

Remember Me

Login

No account?

Register. It's free!

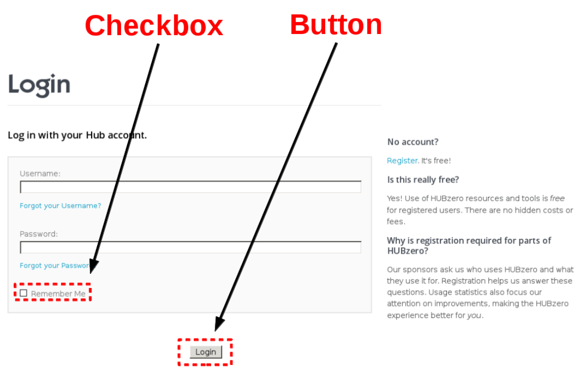
Is this really free?

Yes! Use of HUBzero resources and tools is *free* for registered users. There are no hidden costs or fees.

Why is registration required for parts of HUBzero?

Our sponsors ask us who uses HUBzero and what they use it for. Registration helps us answer these questions. Usage statistics also focus our attention on improvements, making the HUBzero experience better for *you*.

(b) Link widgets



Login

Log in with your Hub account.

Username:

Forgot your Username?

Password:

Forgot your Password?

Remember Me

Login

No account?

Register. It's free!

Is this really free?

Yes! Use of HUBzero resources and tools is *free* for registered users. There are no hidden costs or fees.

Why is registration required for parts of HUBzero?

Our sponsors ask us who uses HUBzero and what they use it for. Registration helps us answer these questions. Usage statistics also focus our attention on improvements, making the HUBzero experience better for *you*.

(c) Checkbox and Button widgets

Fig. 7.3.: The login web page is made up of several types of widgets.

The Login page object has methods that mirror the services offered by the login web page. These methods abstract away the steps required to perform the service and reduce the service to a single call to the page object's API. Listing 7.5 declares four methods that correspond to the services provided by the login page, including login_as(), goto_remind(), goto_reset(), and goto_register().

Listing 7.5: The Login object's methods match the services provided by the widget.

```
1  class Login(BasePageWidget):
2      ...
3      def login_as(self,username,password):
4          self.username.value = username
5          self.password.value = password
6          self.submit.click()
7      def goto_remind(self):
8          self.remind.click()
9      def goto_reset(self):
10         self.reset.click()
11     def goto_register(self):
12         self.register.click()
```

Inside the methods, page object data members representing the web page elements take over to do the real work of typing values into the login form's fields, clicking links or buttons, and toggling checkboxes. For the login_as() method, the self.username and self.password objects are responsible for updating the username and password fields on the web page by using the method's arguments as inputs. self.username and self.password are both instances of the Text class, with a value property which acts as an accessor, allowing users to query or set the current value of the field on the web page. In line 4 of Listing 7.5, self.username's value property is assigned a new value. This assignment causes a Selenium WebDriver handle for the text input field to be retrieved from the web page's HTML DOM. The text input field is cleared of any previous value, and the new value is written to the field on the web page. This happens again, in line 5, for the self.password object. Lines 9 - 12 of Listing 7.6 show the sequence of events in more detail.

Listing 7.6: The Text object manages finding and writing to the Selenium WebDriver object handle.

```
class Text(BasePageWidget):
    @property
    def value(self):
        e = self.owner.find_element_in_owner(self.locator)
        return e.get_attribute('value')

    @value.setter
    def value(self, val):
        e = self.owner.find_element_in_owner(self.locator)
        e.click()
        e.send_keys(Keys.CONTROL,'a')
        e.send_keys(val)
```

7.2.2 Specifying Element Locators for Page Object Classes

The `Login` page object is almost complete. In Listing 7.4, we specified the elements available on the login web page. In Listing 7.5 we added functions to fulfill all of the services offered by the web page. The next step is to specify the locators for the web page elements.

HUBcheck page object classes have a complementary set of locator classes that specify sets of element locators. Listing 7.7 shows two example locator classes for the hubzero.org and nanohub.org login pages discussed in Section 7.1.

Listing 7.7: Locators for the Login page object

```
class Login_Locators_Base_1(object):
    locators = {
        'username'  : "css=#username",
        'password'  : "css=#passwd",
        'submit'    : "css=[name='Submit']",
        ...
    }

class Login_Locators_Base_2(object):
    locators = {
        'username'  : "css=#username",
        'password'  : "css=#password",
        'submit'    : "css=#login-submit",
        ...
    }
```

Each locator class holds a dictionary of locators that is dynamically loaded by the page object class during initialization and is used to override the locators of its data members. In Listing 7.4, the `self.submit` data member is instantiated with the dictionary {`'base': 'submit'`} as one of the arguments. This is a locator override, which maps the `'base'` locator inside of the `self.submit` object to the value of the `'submit'` locator of the `Login` page object. The value of the `'submit'` locators is dynamic because it depends upon which locator class is loaded by the page object, `Login_Locators_Base_1` or `Login_Locators_Base_2`.

The pattern of separating locators from the page object allows HUBcheck page objects to be used across multiple hubs. When new hubs are created new locators can be added to the system, if needed, leaving the page object classes untouched. More often then not, new hubs use locators that have already been added to the HUBcheck library. When new versions of the HUBzero software are released, the locators also can be added to the HUBcheck library without modifying the existing page objects. In the next section, we dive deeper into patterns that can be used to build page objects. While these patterns can be found throughout the HUBcheck library, they include ideas that are best practices for building all kinds of page objects.

7.3 Incorporating Classic Design Patterns into Page Objects

The social aspect of the hub website encourages members to contribute content and organize in communities. To do this, the user must interact with hub web pages containing web forms, evaluate search results represented as lists or tables, and upload content. These three forms of interaction often show up in the web components built for the hub, and by extension, in the page objects built to interface with those web components.

For most web pages, identifying the widgets and services for the page is straight forward. For other web pages, the widgets are not so obvious and inefficient solutions can lead to bad interfaces that add work for developers instead of reducing work. Three types of interactions found on the hub fall into this latter group of web pages: using web forms, reading lists of data, and interacting with items in iframes.

In this section, we'll introduce three patterns, the WebForm pattern, the ItemList pattern, and the IframeWrap pattern, that can be used to quickly produce page objects that work with components that ask the user to interact through web forms, lists of items, or through an iframe.

7.3.1 WebForm Pattern

Web forms are ubiquitous across the web. They are the best way to collect information from the user, so there is no wonder why they are a part of many hub web components. On the hub, users experience them as a part of larger processes to create, update, or delete content. Interacting with most web forms takes place in two phases, population and submission. The first phase, populating the form, involves searching for fields in the form and assigning them values. The second phase, submitting the form, simply involves clicking the submit button on the form.

Many of the harder to test aspects of a hub's web site involve working through multi-step processes that often include web forms. The novice approach to creating page objects for these web forms can lead to unintuitive interfaces. The hub login page and new support ticket page are classic examples of web forms. When evaluated separately, the interfaces may seem very different. Certainly the services provided by the pages are different and so are the fields that need to be populated, but these things can be considered configuration steps for a class that tackles the larger problem of form population and submission.

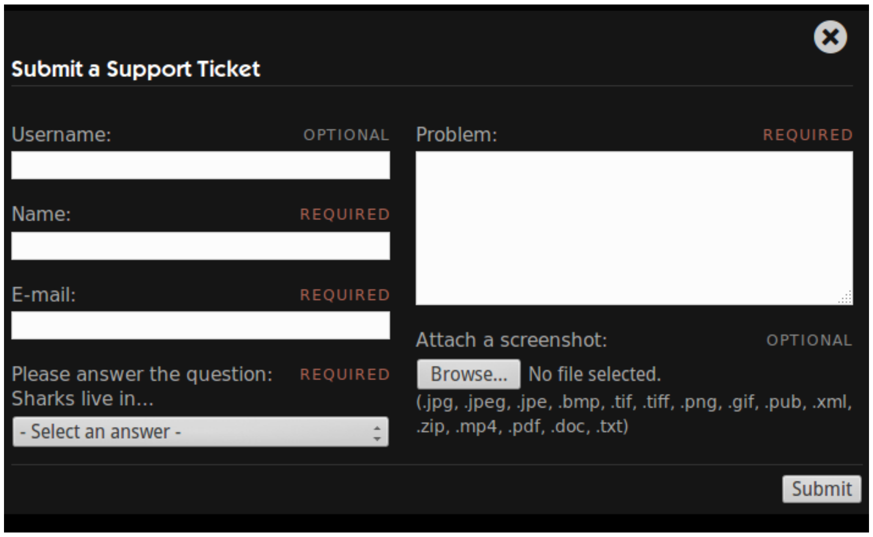

Submit a Support Ticket
Username: OPTIONAL
Problem: REQUIRED
Name: REQUIRED
E-mail: REQUIRED
Attach a screenshot: OPTIONAL
Browse... No file selected.
(.jpg, .jpeg, .jpe, .bmp, .tif, .tiff, .png, .gif, .pub, .xml, .zip, .mp4, .pdf, .doc, .txt)
Please answer the question: REQUIRED
Sharks live in...
- Select an answer -
Submit

Fig. 7.4.: Testing the hubzero.org support ticket form.

Listing 7.8: The typical interface for a web form requires the automation developer to call page object accessors to populate the form.

```
po = TroubleReportForm()

po.set_name('testuser')
po.set_email('tu@hubzero.org')
po.set_problem('test problem')
po.set_upload('myscreen.png')

po.submit.click()
```

The typical page object interface for a web form requires the automation developer to call page object accessor methods to populate each field of a form. Using accessor methods provides a clean interface for the populate phase. An alternative approach is to pass the form inputs to a single method, and allow the method to perform the form population. Similarly, for submitting a web form, a single method can be used to populate and submit the form. This is the idea behind the WebForm pattern. Listing 7.9 shows an example of the interface.

Listing 7.9: An example interface for a web form utilizing two methods, populate_form() to handle filling in the form inputs, and submit_form() to handle form submission.

```
po = TroubleReportForm()

data = {
  'name'    : 'testuser',
  'email'   : 'tu@hubzero.org',
  'problem' : 'test problem',
  'upload'  : 'myscreen.png'
}

po.populate_form(data)

po.submit_form()
```

The purpose of the WebForm pattern is to help standardize the interface for filling out web forms. The usual interface makes the script writer work hard to remember the accessor methods for the inputs on the form. The WebForm pattern encourages the automation developer to organize the inputs for the form and send the inputs to a standard page service, the `populate_form()` method. Later the user can submit the form using another standard service, the `submit_form()` method. These two services are supported for all forms.

Implementing the WebForm pattern requires a page object that represents a web form to be derived from an abstract base class that provides the `populate_form()` and `submit_form()` methods. Listing 7.10 shows an example of such a class.

Listing 7.10: `FormBase` is an example base class for web forms

```
class FormBase(BasePageWidget):
    def __init__(self, owner, locatordict={}):
        ...
        self.submit = Button(self,{'base':'submit'})

    def populate_form(self, data):
        for (k,v) in data:
            widget = getattr(self,k)     # find the widget
            widget.value = v             # set its value

    def submit_form(self,data=[]):
        self.populate_form(data)
        return self.submit.click()
```

In Listing 7.10, the `FormBase` class provides a submit button object in the constructor, and relies on the derived class to supply the fields of the form as data members. The two services of the web form, populating the form and submitting the form, are performed by the class's `populate_form()` and `submit_form()` methods.

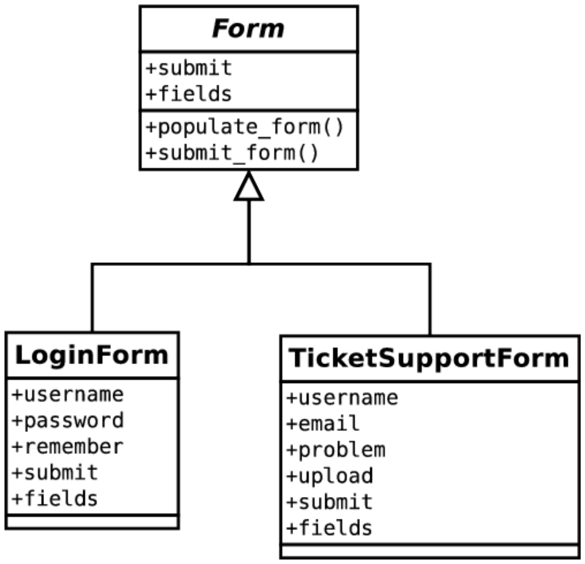


Fig. 7.5.: In the WebForm pattern, the `FormBase` base class provides the two services essential to all web forms, populating the form and submitting the form.

The `FormBase` class can be applied to the hub login page example from Section 7.2 just as easily as it can be applied to the new support ticket page from Figure 7.4, they are both web forms after all. To build a new page object for the login page, the first step is to subclass `FormBase` and add the form's fields to the new class's constructor. Listing 7.11 shows what the new page object class would look like.

Listing 7.11: An example page object for the hub login page, using `FormBase`

```
class LoginPage(FormBase):
    def __init__(self, owner, locatordict={}):
        super(LoginPage,self).__init__(owner,locatordict)
        ...
        self.username = Text(self,{'base':'username'})
        self.password = Text(self,{'base':'password'})
        self.remember = Checkbox(self,{'base':'remember'})
```

The new LoginPage class is derived from the FormBase class. In the LoginPage constructor, we add data members for the fields of the login page's form, the username field, the password field, and the remember checkbox. That's it! Users can interact with the LoginPage page object by supplying the values for the fields as a dictionary or a list of tuples to either of the inherited service methods, `populate_form()` or `submit_form()`, and the service methods take care of inspecting the LoginPage object for the widgets.

Listing 7.12: Interacting with the new LoginPage page object

```
form_data = {'username' : 'testuser',
             'password' : 'abc123',
             'remember' : False}
po = LoginPage()
po.populate_form(form_data.items())
po.submit_form()
```

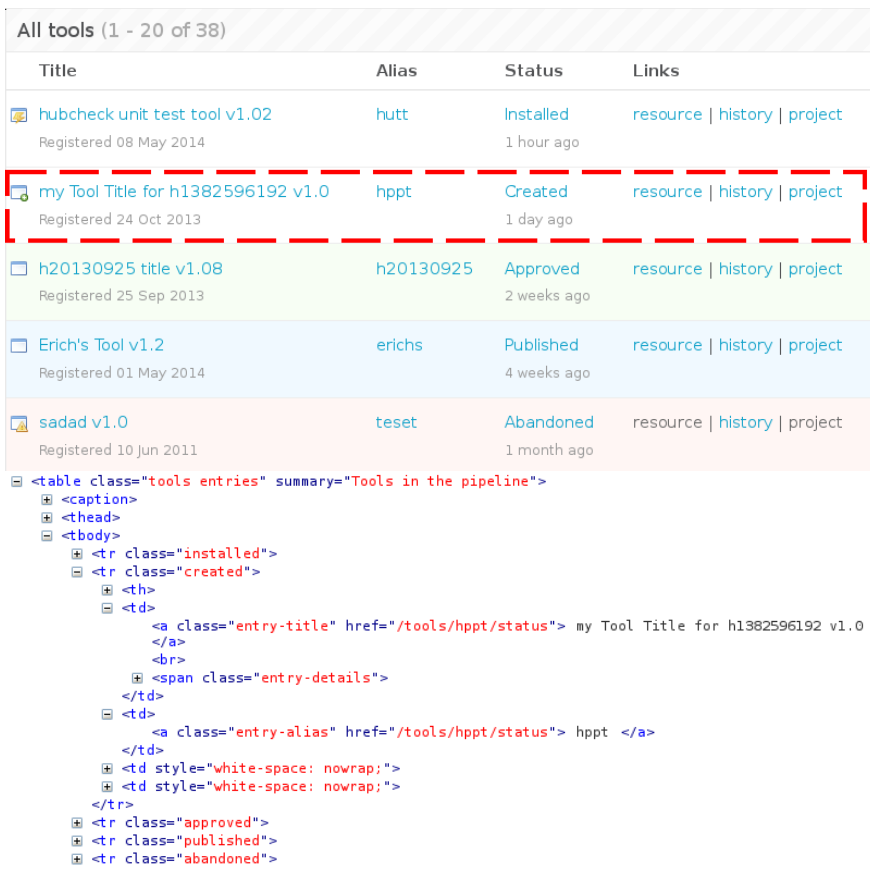
Using the WebForm pattern simplifies building web form page objects to a single step of declaring the fields of the form. There are several variations of the FormBase class, including those that fill out forms in a specified order, validate inputs, or have extra HTML buttons like cancel or preview. These variations can usually be accommodated by adjusting the derived class or the data structures being passed to the base class's service methods.

7.3.2 ItemList Pattern

Lists of items on web pages are often built dynamically, pulling information from databases to present the user with up-to-date information. The amorphous nature of a web page with a dynamically generated list makes it hard to build a static page object to represent it. Since the number of items in the list may fluctuate, it cannot be hard coded into the page object. Even if it could, creating individual page objects for each row/item in the list at once may be memory and time intensive.

Lists of items are an area where the traditional approach to building page objects is hard to apply. Without knowing ahead of time the number of items in the list, static page objects cannot be built with an object representing each item. Instead of manually creating an object for each item in the list, it is better to step back and evaluate the ways lists are used. Lists are a quick, organized way of presenting enumerable pieces of data back to the user. Many lists hold items that provide an overview of the available data by using text fields or links to web pages with more details. Other types of lists contain items that exhibit all available data in a single text field. Regardless of the way the data is displayed, the information in each item depends upon the type of list being presented. Users may interact with a single list item at a time or they may interact with the list as a whole, accessing list properties like the item count and searching for specific items.

The ItemList design pattern defines a way to dynamically instantiate a page object that represents a single, specific item from an arbitrarily sized list of items. Page objects built using this pattern can be used to access the properties of the single item it was instantiated for and can be quickly updated to reference another item in the list. The pattern uses elements of the Iterator pattern [45] and Factory Method pattern [45], and introduces the concept of the *locator template*, a template that has a value substituted into it, to create a new web page element locator.



All tools (1 - 20 of 38)

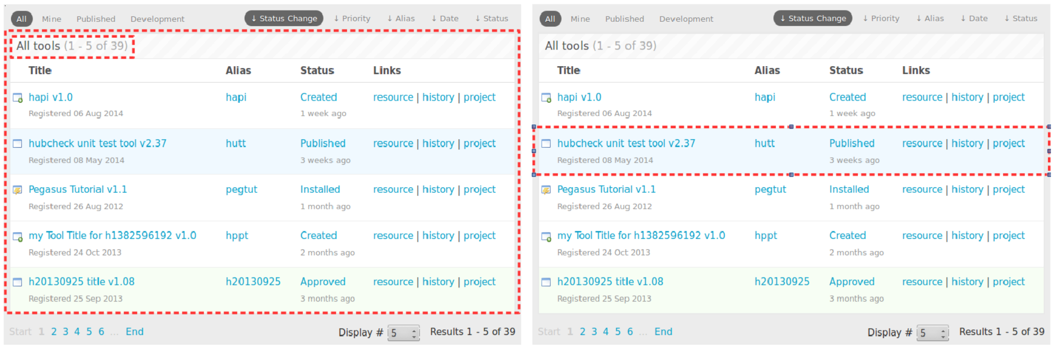
Title	Alias	Status	Links
hubcheck unit test tool v1.02 Registered 08 May 2014	hutt	Installed 1 hour ago	resource | history | project
my Tool Title for h1382596192 v1.0 Registered 24 Oct 2013	hppt	Created 1 day ago	resource | history | project
h20130925 title v1.08 Registered 25 Sep 2013	h20130925	Approved 2 weeks ago	resource | history | project
Erich's Tool v1.2 Registered 01 May 2014	erichs	Published 4 weeks ago	resource | history | project
sadad v1.0 Registered 10 Jun 2011	teset	Abandoned 1 month ago	resource | history | project

```
⊟ <table class="tools entries" summary="Tools in the pipeline">
    ⊞ <caption>
    ⊞ <thead>
    ⊟ <tbody>
        ⊞ <tr class="installed">
        ⊟ <tr class="created">
            ⊞ <th>
            ⊟ <td>
                  <a class="entry-title" href="/tools/hppt/status"> my Tool Title for h1382596192 v1.0
                  </a>
                  <br>
                ⊞ <span class="entry-details">
              </td>
            ⊟ <td>
                  <a class="entry-alias" href="/tools/hppt/status"> hppt  </a>
              </td>
            ⊞ <td style="white-space: nowrap;">
            ⊞ <td style="white-space: nowrap;">
          </tr>
        ⊞ <tr class="approved">
        ⊞ <tr class="published">
        ⊞ <tr class="abandoned">
```

Fig. 7.6.: The hub Tool Pipeline table is a dynamically created list of items. Each row in the table provides links and information regarding a specific simulation tool registered on the hub.

In the hub's tool contribution process, the Tool Pipeline, shown in Figure 7.6, is an HTML table that shows all tools that have been registered on a hub. Each row of the table shows the tool's title, alias, status, time since the tool was registered, time since the last status change, and links leading to the tool's resource, status, and communications history web pages. The Tool Pipeline web page also allows users to search for specific tools by alias.

Building a page object for a dynamic web page like this is hard because the information in the table is pulled from a database and may change over time. While a human user can look at each item quickly to find the item they are interested in, an automated system must iteratively scan all items until matching criteria is found. This pattern of listing information from databases is not unique to the Tool Pipeline page. On the hub, it is also found in the Tags component when displaying all available tags, in the Support component when listing support tickets, the Groups component, Questions and Answers, Projects, Wish List and more.



All tools (1 - 5 of 39)

Title	Alias	Status	Links
hapi v1.0 Registered 06 Aug 2014	hapi	Created 1 week ago	resource | history | project
hubcheck unit test tool v2.37 Registered 08 May 2014	hutt	Published 3 weeks ago	resource | history | project
Pegasus Tutorial v1.1 Registered 26 Aug 2012	pegtut	Installed 1 month ago	resource | history | project
my Tool Title for h1382596192 v1.0 Registered 24 Oct 2013	hppt	Created 2 months ago	resource | history | project
h20130925 title v1.08 Registered 25 Sep 2013	h20130925	Approved 3 months ago	resource | history | project

Start 1 2 3 4 5 6 ... End Display # 5 Results 1 - 5 of 39

(a) Container class coverage

All tools (1 - 5 of 39)

Title	Alias	Status	Links
hapi v1.0 Registered 06 Aug 2014	hapi	Created 1 week ago	resource | history | project
hubcheck unit test tool v2.37 Registered 08 May 2014	hutt	Published 3 weeks ago	resource | history | project
Pegasus Tutorial v1.1 Registered 26 Aug 2012	pegtut	Installed 1 month ago	resource | history | project
my Tool Title for h1382596192 v1.0 Registered 24 Oct 2013	hppt	Created 2 months ago	resource | history | project
h20130925 title v1.08 Registered 25 Sep 2013	h20130925	Approved 3 months ago	resource | history | project

Start 1 2 3 4 5 6 ... End Display # 5 Results 1 - 5 of 39

(b) Item class coverage

Fig. 7.7.: The Container and Item classes are the foundation of the ItemList pattern.

The ItemList pattern provides a standard way to describe and traverse items in a list or table on a web page. The main participants of the pattern are two base classes, `Container` and `Item`. The container class represents the meta-data of the list. Automation developers can ask the container questions regarding properties of the list, like *how many items are in the list?* The container cannot answer questions about specific items in the list, but can provide access to the list elements, either sequentially or through a limited search capability. The item class represents a single item in the list. Automation developers can query this class for information about the item it represents in the list. Additionally, the item class can be updated on the fly to point to another item from the list. The container and item classes provide interfaces that, when implemented, can be used to represent lists, tables, and other data structures that appear on the web page as a collection of items.

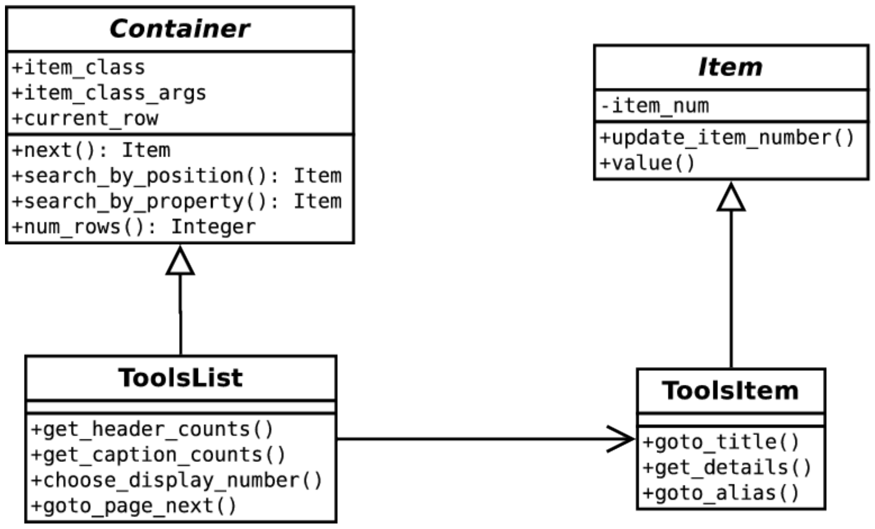


Fig. 7.8.: The ItemList pattern uses a container class to represent the list and provide access to list meta-data while providing access to elements of the list through an item class. It incorporates the Iterator and FactoryMethod patterns.

We can further explore the ItemList pattern by building example page objects to represent the Tool Pipeline table found in the Tools component on the hub and shown in Figure 7.6. As mentioned earlier, the Tool Pipeline is an HTML table where each row holds the details and links of a tool resource that has been registered on the hub. A single page object class that represents the whole table would be difficult to manage due to the dynamic nature of the information being displayed. Alternatively, the table can be easily represented by two smaller classes, a container class named `ToolsList` and an item class named `ToolsItem`.

The `ToolsItem` class represents an item in the Tool Pipeline table, as shown in Figure 7.7b. It is an `Item` class, and as such, allows the automation developer to query details about the specific item it represents such as the tool's title, alias, and status, and provides access to the web page links that host more information about the tool. It also provides a way to update which item the object represents in the list. In the `ToolsItem` class, these features are provided by the `value()` and `update_item_number()` methods.

Listing 7.13: The Item class interface, implemented by the ToolsItem class.

```
class Item(BasePageWidget):
    def __init__(self, owner, locatordict, item_number):
        ...

    def value(self):
        ...

    def update_item_number(self,item_number):
        ...
```

The `ToolsItem` class's constructor, outlined in Listing 7.13, accepts a dictionary of web element locator templates in the `locatordict` parameter. Locator templates [46] are different from regular locators, previously discussed in Section 7.2.2, in that they can have values substituted into them.

Listing 7.14: Locator templates allow values to be substituted into them.

```
    locators = {
        'title'    : "css=tr:nth-of-type({item_num}) .title",
        'details'  : "css=tr:nth-of-type({item_num}) .details",
        'alias'    : "css=tr:nth-of-type({item_num}) .alias",
        'status'   : "css=tr:nth-of-type({item_num}) .status",
        'time'     : "css=tr:nth-of-type({item_num}) .time",
        'resource' : "css=tr:nth-of-type({item_num}) .page",
        'history'  : "css=tr:nth-of-type({item_num}) .history",
        'wiki'     : "css=tr:nth-of-type({item_num}) .wiki",
    }
```

Listing 7.14 shows an example dictionary of locator templates for the `ToolsItem` class. The templates contain a `item_num` placeholder, which is substituted with a real value, the `item_number` parameter from the ToolsItem constructor, when the ToolsItem object is instantiated. The value substituted into the template can also be changed by calling the object's `update_item_number()` method which changes the value substituted into the template locators and requests children page objects to update their locators, propagating the change through the page object hierarchy.

Listing 7.15: The `update_item_number()` method updates the item being referenced in the list

```python
    def update_item_number(self,item_number):
        self._item_number = item_number
        # format all locator templates
        for k,v in self.locators.items():
            self.locators[k] = v.format(item_num=self._item_number)
        # update this object's children
        self.update_locators_in_widgets()
```

Listing 7.16: The `ToolsItem` class's `__init__()` method describes the components of a single item, the widgets in the item the automation developer would want to interact with.

```python
class ToolsItem(Item) :
    ...
    def __init__(self, owner, locatordict, item_number):
        ...
        self._item_number  = item_number
        self.title        = Link(self,{'base':'title'})
        self.details      = TextReadOnly(self,{'base':'details'})
        self.alias        = Link(self,{'base':'alias'})
        self.status       = Link(self,{'base':'status'})
        self.time         = TextReadOnly(self,{'base':'time'})
        self.resource     = Link(self,{'base':'resource'})
        self.history      = Link(self,{'base':'history'})
        self.wiki         = Link(self,{'base':'wiki'})
        ...
```

The items of the Tool Pipeline table have eight components, five of which can be considered properties, including title, register details, alias, status, and time since status change. The other three components, resource link, history link, and wiki link, are links to web pages about the resource. The value of the item can be described as a collection of the item's properties. The `value()` method provides access to the item's properties, through the dictionary it returns.

Listing 7.17: The `ToolsItem` class's `value()` method returns a dictionary of property values

```
class ToolsItem(Item) :
    ...
    def value(self):
        """return a dictionary of properties for this item"""

        properties = {
            'title'     : self.title.text(),
            'details'   : self.details.value,
            'alias'     : self.alias.text(),
            'status'    : self.status.text(),
            'time'      : self.time.value,
        }

        return properties
```

The `ToolsList` class represents the Tool Pipeline table as a container of items as shown in Figure 7.7a. As a `Container` class, `ToolsList` provides the automation developer with the ability to interact with the features of the table that are independent of specific items, like retrieving the counts from the table's caption, getting the number of tools displayed, searching for tools by name, and iterating over all of the displayed tools. Listing 7.18 shows an outline of the `Container` class, the basis for `ToolsList`.

Listing 7.18: The `Container` class interface, implemented by the `ToolsList` class, providing automation developers with access to the Tool Pipeline table meta-data and items

```
class Container(BasePageWidget):
    def __init__(self, owner, locatordict):
        ...
    def __iter__(self):
        ...
    def next(self):
        ...
    def get_item_by_position(self,item_number):
        ...
    def get_item_by_property(self,prop,val,compare=None):
        ...
    def num_items(self):
        ...
    def header_counts(self):
        ...
```

The `ToolsList` class provides sequential access to the items of the container through an iterator by implementing the Iterator pattern. The goal of the Iterator pattern is to allow users to sequentially access elements of the collection without knowing anything about the underlying structure of the elements or the collection. The pattern is usually associated with collections containing elements of different types, but works equally well when the elements are homogeneous. Essentially, the pattern allows the automation developer to keep asking for the next element and the container keeps returning new elements from the collection until there are no new elements to return.

Python helps us create iterator objects by providing an Iterator protocol. In Python, classes that define the `__iter__()` method can return an iterator object. The iterator object needs to define the `next()` method, which provides the caller an element from the collection being iterated over. The `Container` class, `ToolsList`, can act as an iterator by defining the `__iter__()` and `next()` methods. The only

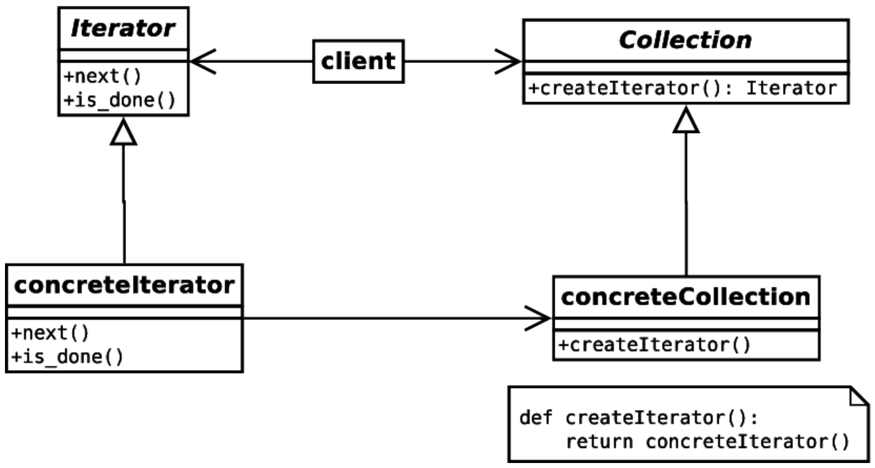


Fig. 7.9.: Containers implement the Iterator pattern to allow sequential access to items.

extra bit it needs to do is to keep track of the current item, which it does through the `__current_item` variable, shown in Listing 7.19.

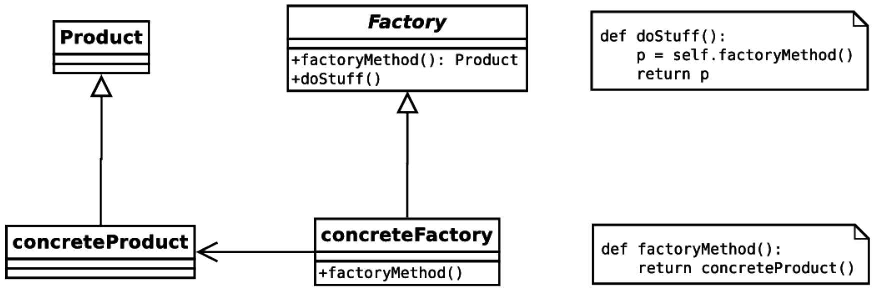
Listing 7.19: The `Container` class implements the Iterator pattern to provide sequential access to items.

```python
def Container(BasePageWidget):
    ...
    def __iter__(self):
        self.__current_item = 0
        return self

    def next(self):
        ...
        self.__current_item += 1

        if self.__current_item >= self__num_items:
            # reset our counter, stop iterating
            self.__current_item = 0
            raise StopIteration

        return self.get_item_by_position(self.__current_item)
```

The last task of a `Container` class is to provide a way to search the list for items that match some criteria. Two popular ways to access items from the container are by position and by property. The type of item returned by the search methods is tied to the container returning it. For example, a container designed to represent a HUBzero support ticket list will return support ticket list items and a container designed to represent a HUBzero group list will return group list items.

To keep the `Container` class generic, it implements the Factory Method pattern, allowing subclasses to define which `Item` class to instantiate.

The Factory Method Pattern is used when we want to define an interface for creating an object, but don't know which class to instantiate. Instead of always instantiating the same class, we delegate the responsibility to a subclass.



```
def doStuff():
    p = self.factoryMethod()
    return p
```

```
def factoryMethod():
    return concreteProduct()
```

Fig. 7.10.: Containers use the Factory Method pattern to allow derived classes determine the type of Item class to return from searches.

In the case of the `Container` class and the Tool Pipeline example, to be able to return `ToolsItem` objects from searches the `ToolsList` object must know how to create them. The `Item` class and parameters to create an object are saved by the derived container, and then used by the search methods to instantiate the correct `Item` objects for the container.

Listing 7.20: The `ToolsList` class manages which type of `Item` object to return from searches. In this case `ToolsItem` objects.

```python
class Container(BasePageWidget):
    def __init__(self, owner, locatordict):
        self.item_class = None
        self.item_class_args = None

class ToolsList(Container):
    def __init__(self, owner, locatordict):
        ...
        self.item_class = ToolsItem
        self.item_class_args = [{...}]
```

The `ToolsList` class provides two methods for searching through a list of items. The first method, `get_item_by_position()`, allows automation developers to search for an item in the list by position. For example, developers can ask for the fifth item in the list. Listing 7.21 shows an implementation of this method that accepts a counting parameter, `item_number`, representing the n-th item in the list. The method uses the `Container` object's `Item` class variable, `__item_class`, to construct the `Item` object for the n-th item in the list. `__item_class` uses the `Item` class parameters, stored in `__item_class_args`, and the method's `item_number` parameter to configure the new page object representing the specific item.

Listing 7.21: The Container class can retrieve Items by position

```python
    def get_item_by_position(self,item_number):
        result = self.__item_class(
                    self.owner,
                    *self.__item_class_args,
                    item_number=item_number)
        result.detach_from_owner()
        return result
```

The second method, `get_item_by_property()`, allows the developer to search for the first item in the list that matches a property constraint. The method accepts two required parameters, the name of the property and the value it should match.

Listing 7.22: The Container class can retrieve Items by property

```
    def get_item_by_property(self,prop,val):
        result = None

        # create a default item object, using the first item
        r = self.__item_class(self.owner,*self.__item_class_args,item_number=1)
        r.detach_from_owner()

        for item_number in xrange(1,len(items)+1):

            # update the default item object to point to the current item
            r.update_item_number(item_number)

            # check if our current item matches the property constraint
            if r.value()[prop] == val:
              result = r
              break

        # if no items matched, clean up our default item object
        if result is None:
          del r

        return result
```

The `get_item_by_property()` method also uses the `__item_class` variable, representing the `Item` class, to construct a new page object that represents a single item in the list. Similar to the `get_item_by_position()` method, the `get_item_by_property()` method passes the `__item_class`'s constructor a list of arguments to configure the new page object, including a dictionary of locator templates.

The `get_item_by_property()` method uses an `Item` object to iterate over the items in the list, searching for the first item that satisfies the property constraint. It takes advantage of the `Item` object's `update_item_number()` method and template locators to update the locators of the object without instantiating a new page object for each item it encounters in the list. In Listing 7.22, you can see the

`Item` object being updated inside of the for loop in line 11, and the comparison between the item's property and the requested value in line 14.

The ItemList pattern focuses on the interactions of two classes, the `Container` class and the `Item` class. In the Tool Pipeline table example the `Container` class, `ToolsList`, was responsible for all of the services provided by the table that were not related to a specific item or item in the table. The `Item` class, `ToolsItem` was responsible for services associated with a specific item or item in the table. This separation of services, along with the use of web element locator templates, allowed the `Container` class to dynamically create page objects for specific items in the table and update the item being referenced without needing to destroy and create a new object.

7.3.3 IframeWrap Pattern

Iframes are another area where using the wrong design can make page objects difficult to build and inefficient to use and maintain. When hub users upload resources to a HUBzero website, they are asked to respond to several questions on a web form, one of which involves describing the resource they are contributing. In previous incarnations, the resource contribution form's description field was a simple HTML <textarea> element. The field handled both plain text descriptions, but also allowed users to enter a wiki-like markup language that produced rich text descriptions. Building a page object with a <textarea> is pretty rudimentary, and in HUBcheck is represented by the `TextArea` class.

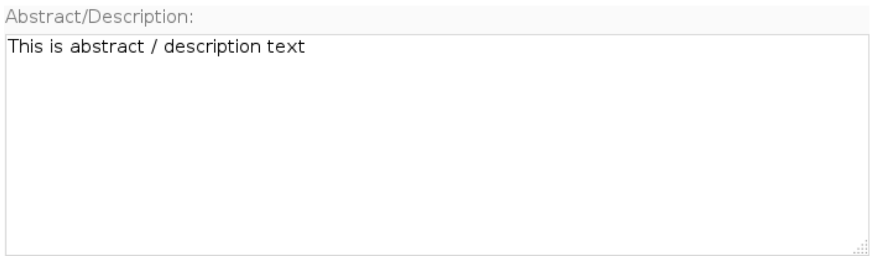
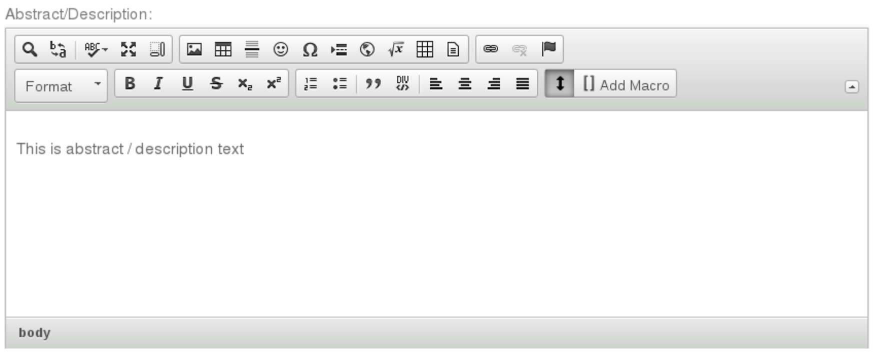

Abstract/Description:
This is abstract / description text

Fig. 7.11.: HTML <textarea> based editor.

Listing 7.23: HTML of the textarea based editor.

```html
<label for="field-fulltxt">
    Abstract/Description:
    <textarea id="field-fulltxt">This is abstract / description text</textarea>
</label>
```

Around the release of version 1.2.0 of the HUBzero software, the web developers started incorporating a new editor for the description field. Shown in Figure 7.12, the new editor incorporated better controls for handling rich text. Instead of writing out the wiki syntax to make words bold, for example, the hub user would press the bold button in the editor and type the text they wanted to be bold. This was a great step forward for usability. The updated editor meant an update was needed for the page objects that interacted with the previously available <textarea> based editor. Such a drastic widget change like this usually results in a new page object being created.



This is abstract / description text

Fig. 7.12.: HTML iframe based editor.

Listing 7.24: HTML of the iframe based editor, where writing to the <body> tag is just like writing to the <textarea> tag after the automation script enters the iframe.

```html
<iframe class="cke_wysiwg_frame">
    <html>
        <body class="ckeditor-body">
            <p>This is abstract / description text </p>
        </body>
    </html>
</iframe>
```

Investigating the new web page, one could see that the previous <textarea> element was replaced with an iframe and embedded web page. Playing around with the iframe element revealed that once the automation script entered the iframe, writing text to the <body> element was just like writing text to the <textarea> element. This raised the question:

> *Do I need to write a new page object class for an element embedded in an iframe, if a class for that element already exists and works with the exception of entering and exiting the iframe?*

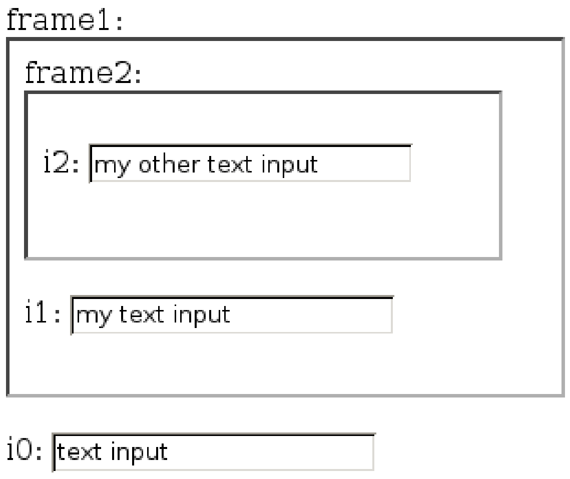


Fig. 7.13.: Text input fields embedded in different levels of iframes.

Before approaching this question, let's first investigate how iframes work. Figure 7.13 shows an example web page with some input fields embedded in different levels of iframes.

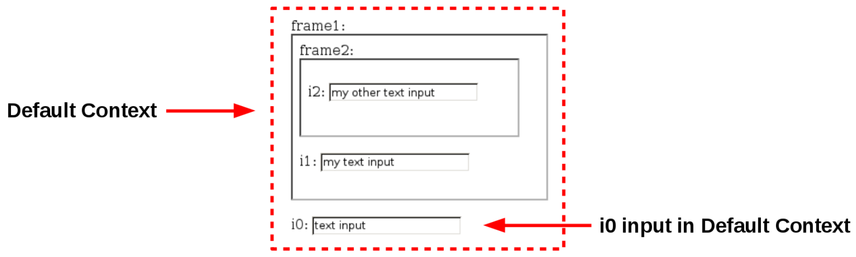


Fig. 7.14.: Text input i0 exists in the default context.

Listing 7.25: In the default context, iframe frame1 references inner_page.html.

```
<html>
    <body>
        <label for="frame1">frame1: </label>
        <iframe id="frame1" src="inner_page.html"></iframe>
        <br/><br/>
        <label for="i0">i0: </label>
        <input type="text" id="i0" value="text input"></input>
    </body>
</html>
```

The first input field, i0, is located in the default context. This is the level of the web page we generally work in when iframes are not involved. Along with the input field i0, this web page also has an iframe, frame1. In the HTML, iframes hold the location of another web page to be embedded in the frame. In this example, frame1 is going to load up the web page inner_page.html.

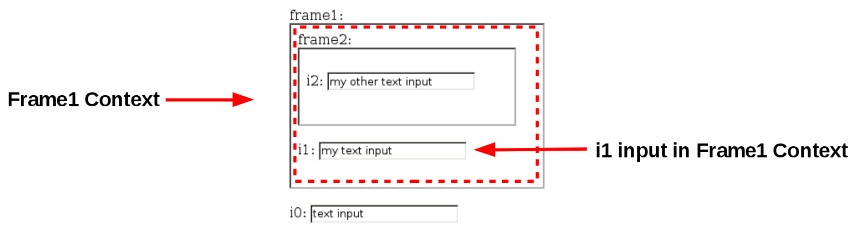


Fig. 7.15.: Text input i1 exists in the frame1 context.

Listing 7.26: inner_page.html - frame2 references another_page.html.

```
<html>
    <body>
        <label for="frame2">frame2: </label>
        <iframe id="frame2" src="another_page.html"></iframe>
        <br/><br/>
        <label for="i1">i1: </label>
        <input type="text" id="i1" value="my text input"></input>
    </body>
</html>
```

Figure 7.15 and Listing 7.26 show the HTML for inner_page.html. It makes up what is referred to as the Frame1 Context. The Frame1 context has an input field i1, and another iframe, frame2. Again, the iframe frame2 holds the location of a web page, and in this case it points to another_page.html.

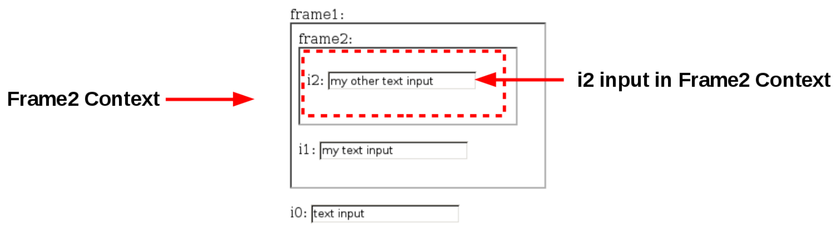


Fig. 7.16.: Text input i2 exists in the frame2 context.

Listing 7.27: another_page.html - in frame2 context, only input i2 exists.

```
<html>
    <body>
        <label for="i2">i2: </label>
        <input type="text" id="i2" value="my other text input"></input>
    </body>
</html>
```

Listing 7.27 shows the HTML for the file another_page.html that makes up the Frame2 Context. It contains an input field i2 that resides within the Frame2 context.

Listing 7.28: Page object class for i0 text input field.

```
class Text(BasePageWidget):
    ...
    # setter
    def value(self, text):
        e = self.find_element(self.locator)
        e.clear()
        e.send_keys(text)
    ...
```

A page object for the i0 input field that resides in the Default Context would probably include a getter method to retrieve the value of the input, a setter method to set the value of the input, and maybe an append method to assist with appending text to whatever was already in the field. Since the i0 field resides in the default context, there is no need to do anything special; the methods will find the i0 input element in the HTML DOM, and perform actions on it.

Listing 7.29: Page object class for i1 text <input> field.

```
 1 class Text1Frame(BasePageWidget):
 2     ...
 3     # setter
 4     def value(self, text):
 5         frame = self.find_element('#frame1')
 6         self._browser.switch_to_frame(frame)
 7         e = self.find_element(self.locator)
 8         e.clear()
 9         e.send_keys(text)
10         self._browser.switch_to_default_content()
11     ...
```

Building a page object for the i1 input field involves a little more work. The page object is almost the same as the one for the i0 input field, but because i1 is located inside of the Frame1 context the web browser needs to be instructed to traverse the frame1 iframe before performing any getter, setter, or append actions. So inside each of the page object's methods a few lines of code need to be added for entering and exiting the iframe. After the web browser has entered the Frame1 context it can search for the element in the HTML DOM and perform actions on the element. In Listing 7.29, the extra code for entering and exiting the Frame1 context shows up in lines 5-6 and line 10. Lines 7-9 make up the core action of the widget and are the same as what is found in the page object for text input i0, in Listing 7.28.

Listing 7.30: Page object class for i2 text input field.

```
1  class Text2Frame(BasePageWidget):
2      ...
3      # setter
4      def value(self, text):
5          frame1 = self.find_element('#frame1')
6          self._browser.switch_to_frame(frame1)
7          frame2 = self.find_element('#frame2')
8          self._browser.switch_to_frame(frame2)
9          e = self.find_element(self.locator)
10         e.clear()
11         e.send_keys(text)
12         self._browser.switch_to_default_content()
13     ...
```

Building a page object for the i2 input field adds another layer of iframe context traversal. Remember, i2 is located inside of the Frame2 context, which is located inside of the Frame1 context. The methods for the i2 page object have the same core actions as those of the i0 and i1 page objects, but include code to traverse two iframe contexts. This can be seen in Listing 7.30, where the setter method first moves from the Default context to the Frame1 context in lines 5-6, then moves from the Frame1 to the Frame2 context in lines 7-8. The setter method next performs the method's core action in lines 9-11 and finally returns back to the default context in line 12.

To review, building page objects for the i0, i1 and i2 input fields involves tracking the current frame level and possibly traversing frame levels to be in the correct context for interacting with an element. In the example from Figure 7.13, all of the page objects started off with the same code, but for input i1, additional lines were added to account for entering and exiting the Frame1 context. Similarly, for input i2, additional lines of code were added to account for entering and exiting the Frame1 and Frame2 contexts. But the question remains:

> *Is there a way to handle the entering and exiting of iframes outside of the page object so we can reuse our original page object that represents a Text <input> field on a web page?*

In essence, a solution would provide a way to write the core methods of a page object once and if the page object was found inside of an iframe, the methods could be wrapped with code to enter and exit the iframe. If the page object was found inside of two, or three, or more iframes, the methods would just keep getting wrapped with code to enter and exit iframe contexts.

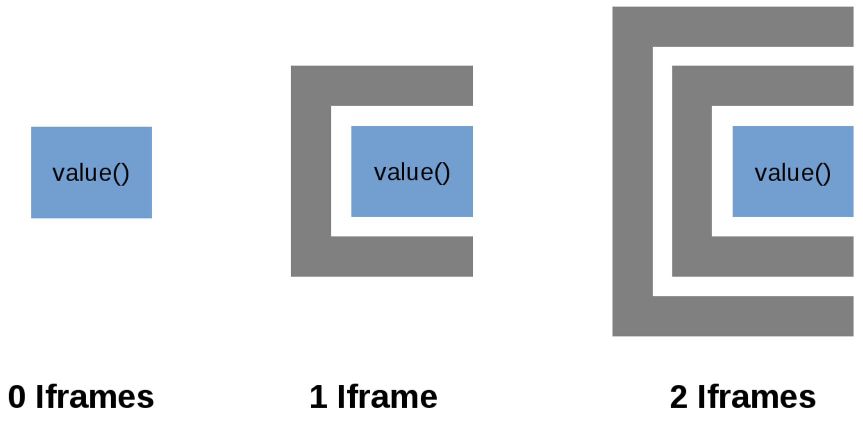


Fig. 7.17.: IframeWrap pattern wraps the core of methods with code to traverse iframe contexts.

This is the idea behind the IframeWrap pattern. It uses the Decorator pattern to *decorate* or wrap the attributes of a page object with code to enter an iframe context, call the original page object method, and then exit the iframe context. It supports both page objects from the default context as well as previously wrapped page objects.

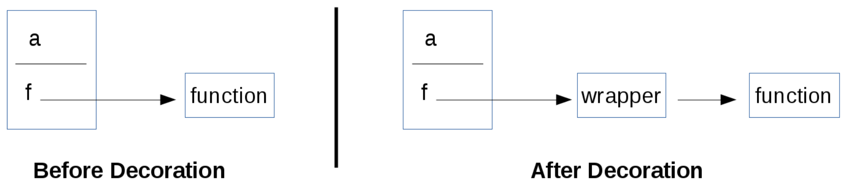


Fig. 7.18.: Decorator pattern

The purpose of the Decorator pattern is to extend functionality of an object without necessarily changing the interface. It is most often used when a specific object needs to be changed at runtime without affecting other objects of the class. It does this by wrapping the original functionality of the object's attributes with code to do extra work.

Consider the example object `a` shown in Figure 7.18, which is an instance of the class `A`, with a method `f`. Additional responsibilities can be added to `a`'s method `f` by first pointing `a.f` to a wrapper method and then directing the wrapper method to perform the extra work and call the method that `a.f` used to point to.

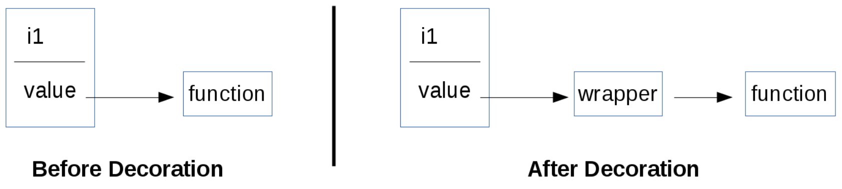


Fig. 7.19.: Decorator pattern applied to text input field i1 in Frame1 context

The same idea can be applied to the `Text` page object, from Listing 7.28, to create a page object for the i1 input in the Frame1 context. The page object's `value()` method performs the core actions for setting or getting the value of the underlying HTML element. In Python, `value()` is an attribute that points to a function object, which can be decorated to add the enter and exit iframe commands. After decorating the function object, the value attribute will point to a wrapper function that enters the correct iframe context, calls the original function object, and returns back to the default context.

Listing 7.31: Page object class for web page with multiple text input field embedded in frames.

```
class FramedInputs(BasePageObject):
    def __init__(self):
        self.i0 = Text('#i0')
        self.i1 = IframeWrap( Text('#i1'), ['#frame1'] )
        self.i2 = IframeWrap( Text('#i2'), ['#frame2', '#frame1'] )
```

A page object class that represents the web page shown in Figure 7.13 can be built by instantiating and decorating the `Text` class from Listing 7.28. In Listing 7.31, the variable self.i0 instantiates a `Text` object to represent the text input field in the default context. The variables self.i1 and self.i2 also instantiate `Text` objects, but immediately passes them to the `IframeWrap()` function. The `IframeWrap()` function accepts a page object and a list of locators for frames that must be traversed to access the element represented by the page object. For example, the i1 text input field resides within a frame with a locator `#frame1`, so the `IframeWrap()` function is sent the single element list `['#frame1']`. Similarly, the i2 text input field resides in a frame with a locator `#frame2`, which itself resides in a frame with a locator `#frame1`, so the `IframeWrap()` function is sent a two element list containing both locators.

Listing 7.32: `IframeTracker` objects manage wrapping object methods and attributes

```
class IframeTracker(object):
    ...
    def wrap_callable_attributes(self,o):
        for attr, item in o.__class__.__dict__.items():
            if callable(item):
                item = getattr(o,attr)
                setattr(o,attr,self.wrap_attribute(item))

    def wrap_attribute(self,item):
        def wrapper(*args, **kwargs):
            ...
            switched = self._switch_to_iframe_context(final_framelevel)
            result = item(*args, **kwargs)
            self._switch_to_iframe_context(initial_framelevel)
            return result
        return wrapper
```

Internally, the `IframeWrap()` function creates an `IframeTracker` object and associates it with the page object that is to be decorated. The `IframeTracker` object is responsible for tracking frame levels and wrapping the object's attributes. The most interesting part of the `IframeTracker` object is when the page object gets decorated, which is shown in Listing 7.32. First the `IframeWrap()` method calls the `wrap_callable_attributes()` method, which identifies callable attributes from the object, including methods. Within the page object, the callable attribute is replaced with a call to the `wrapper()` method, which is a closure that stores the function object it replaces. Just as shown in Figure 7.18, the `wrapper()` method takes care of entering the correct frame context, calling the original method it replaces that performs the core actions, and returning back to the original frame context.

Listing 7.33: IframeWrap'd page objects work just like non-wrapped page objects.

```python
class FramedInputs(BasePageObject):
    def __init__(self):
        self.i0 = Text('#i0')
        self.i1 = IframeWrap( Text('#i1'), ['#frame1'] )
        self.i2 = IframeWrap( Text('#i2'), ['#frame2', '#frame1'] )


po = FramedInputs()

# print out the current text in the widgets
print "i0.value = %s" % (po.i0.value)
print "i1.value = %s" % (po.i1.value)
print "i2.value = %s" % (po.i2.value)

# update the text in the widgets
po.i0.value = 'i0 text'
po.i1.value = 'new i1 text'
po.i2.value = 'new i2 text too'
```

With the page object attribute wrapping process complete, the newly IframeWrap'd page object is ready for use. Inside of automation scripts, the decorated page objects work just like the non-decorated page objects. The automation developer doesn't need to do anything special to access or interact with the wrapped page objects. In Listing 7.33, the variables i0, i1, and i2 are all accessed from the page object po in the same way. The responsibility for managing the frame traversal has been embedded within the page object. The difference between text input field page objects is only exposed in how the page objects are instantiated.

When implementing the IframeWrap pattern in Python there are a few gotchas. Not all page object attributes need to be wrapped. Some attributes do not try to interact with the web element the page object represents. Keeping a list of these attributes is handy so the wrapping of these attributes can be skipped. Additionally wrapping Python object properties can be tricky because properties are a part of the class, not the object. It requires creating a new class object from the page object's class, decorating the properties of the new class object, and associating the page object with the new class.

7.4 Summary of Page Object Based Design Patterns

The Page Object design pattern provides developers with an object-oriented way to think about dissecting web pages. Its framework for creating reusable pieces of code to represent parts of web pages can simplify the process of building robust web automation software, but sometimes taking a naive approach can lead to an inefficient page object design. In this chapter, we saw three situations where implementing a naive page object design could lead to inefficient programs. Instead, more robust designs were offered as solutions which helped increase code reuse.

The first situation involved the frequent situation of building page objects for web forms. Due to similarities in the design and goals of web forms, we were able to reduce their use to answering questions, filling the answers into fields on the web page, and pressing the submit or cancel button. This generalization of goals allowed us to recognize that with the exception of the specific fields that needed to be populated, a large portion of building page objects for web forms could be abstracted into a generic `Form` superclass. Fields could then be specified in a derived subclass.

In the second case we investigated building page objects for lists of items on a web page where there is no a priori knowledge of the number of items displayed. The ItemList Pattern encourages users to build a single page object to represent an item in a list, and update which item in the list the page object points to instead of trying to build a separate class for each item in the list.

Lastly, we considered building page objects for web page elements that exist in different iframe levels. While it may be tempting to manually create a new page object class for an element that exists in an iframe, this approach discourages code reuse. Instead, the IframeWrap pattern decorates a page object's attributes with additional code to enter and exit the proper iframe context without changing the original object's interface.

8. HUBCHECK AS A SOLUTION

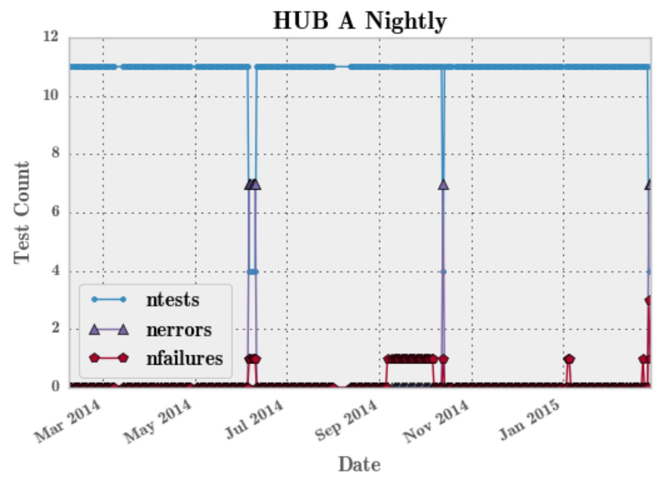
In February of 2014, the HUBzero team deployed a Jenkins [47] continuous integration server to help manage automated tests being run by HUBcheck and to make test results more accessible to members of the team. Since then, automated test results have been tracked for runs against the weekly and nightly test suites, two of the most frequently run test suites offered by HUBcheck. Through the use of the weekly and nightly test suites, the HUBzero team was able to track the health of hubs and resolve differences in their setup as they receive website and tool session container upgrades.

The HUBzero team has tracked the long term results of two of HUBcheck's test suites. The nightly suite contains 11 test cases that focus on tool session container access and job submission through the **submit** command. The weekly suite contains over 150 test cases and dives deeper into the setup and workings of the tool session container, including some of the most tedious tasks to check, which involve interaction between the user's website and tool session container. As their names suggest, the nightly and weekly suites run on a nightly and weekly basis respectively.

The health of a hub can be categorized as either *Turbulent*, *Upgraded*, or *New*. Below, we look at each categorization and use the HUBcheck's weekly and nightly test suites to help identify opportunities for improvement in the roll out, maintenance, and upgrade of hubs.

8.1 Turbulent Hubs

Turbulent hubs are typically older hubs that have gone through a number of software upgrades in the past, when the upgrade process was more relaxed and testing was less of a priority. Over the monitoring period, the HUBcheck test suites were critical in helping the HUBzero team identify and document problems on hubs as



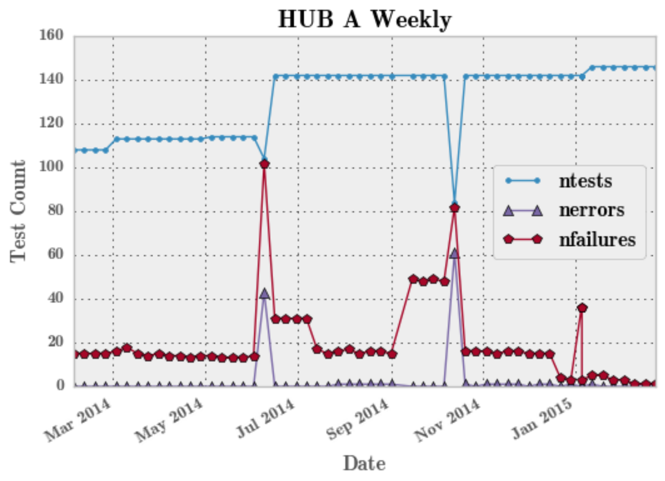


Fig. 8.1.: HUBcheck was used to track the health of hub A, a *Turbulent* hub, as website and tool session container software were upgraded through 2014. **ntests** represents the number of tests in the HUBcheck test suite that ran and completed with either a pass or fail status. **nerrors** represents the number of HUBcheck tests that partially ran and exited due to an exception being raised. In a small number of cases, the exception is related to an error in the HUBcheck library. An overwhelming amount of the time, these errors signal a problem on the hub that prevented the test from being properly setup. **nfailures** represents the number of tests that completed and failed due to an assertion.

software was upgraded and machines were rebooted. The history of one particularly turbulent hub, hub A, can be seen in Figure 8.1.

For hub A, tracking began on February 03, 2014 where, at that time, 15 of the 108 tests in the weekly test suite were failing. The testing and failures were focused on the tool session container configuration. This was an area of active concern for the team because hub A was in the middle of transitioning the simulation tools running in containers using the Debian 6 operating system to containers using the Debian 7 operating system. Between March 2014 and June 2014, the weekly suite continued to identify 15 test failures.

A number of events occurred on hub A that contributed to its fluctuating status. The first manifested itself at the beginning of June 2014 and was somewhat hidden by another event. The more obvious event at the beginning of June 2014 shows up as a spike in test failures and errors in the weekly test suite results. The spike was due to an emergency kernel security patch that required the shutdown of a machine hosting tool session containers for hub A. Hidden in the background of this spike was an increase of 17 additional test failures that were associated with new parameter passing tests that had been added to the weekly test suite that same week. The newly failing tests showed that code to support parameter passing was not available on the hub. This is likely due to the hub needing a website code update. In the beginning of July 2014, a code update was performed and the next run of the weekly test suite showed 15 of the 17 previously failing tests began to pass. The two remaining failing parameter passing tests were identified as bugs in HUBzero's core implementation and fixes for them are being deployed on various hubs.

The second event that elevated test failures occurred at the beginning of September 2014, when a miscommunication lead to a change in the system that was not immediately obvious. Symptoms of the problem caused several HUBcheck tests to fail in the weeks that followed. In the beginning of October 2014, the problem was identified and fixed.

Since October 2014, hub A has shown a decrease in test failures, signaling a healthier hub. When major events do arise, they have been addressed quickly due to rigorous monitoring of HUBcheck. For example, early in January 2015, HUBcheck test failures uncovered a configuration change that occurred after a system reboot. The HUBzero team was able to quickly identify and resolve the problem before hub users were affected. With HUBcheck monitoring, hub A is on its way to becoming like other *Upgraded* hubs.

Turbulent hubs are not common for the HUBzero team. Recently, hubs that would have been classified as turbulent have been taken down as opposed to being put through the website code update process. For the Turbulent hubs that remain in service, HUBcheck's test suites have been used to monitor their health and work towards applying updates for website code and tool container configurations. To keep hubs from falling into this state, the HUBzero team should continue to add tests to the test suites that check hub component functionality.

8.2 Upgraded Hubs

Most hubs have a hub health signature that resembles that of an *Upgraded* hub, where the HUBcheck weekly test suite results start off with a number of test failures that steadily decrease over time. On Upgraded hubs, increases in test failures may be seen at predictable times, such as right after hub updates or during planned maintenance, making them easy to identify and explain on the nightly and weekly suite result graphs.

Hub B was brought online at about the same time as hub A. Prior to February 2014, it would have been identified as a *Turbulent* hub due to its use of vintage website code and tool session container configurations, but today it is an example of an *Upgraded* hub. The hub received a code update just before the monitoring period began, but HUBcheck nightly and weekly suites were still instrumental in helping to identify and fix many of the problems the hub was experiencing before the upgrade. In February 2014, monitoring of hub B reported 23 test failures until the end of April. Throughout April, the HUBzero team focused on reducing test failures on hub B. By the end of April, the number of failures had dropped from 23 to 17 as new tests were introduced into the weekly test suite. This decreasing trend is again seen near the middle of May, when the number of test failures dropped to 4 tests. The weekly

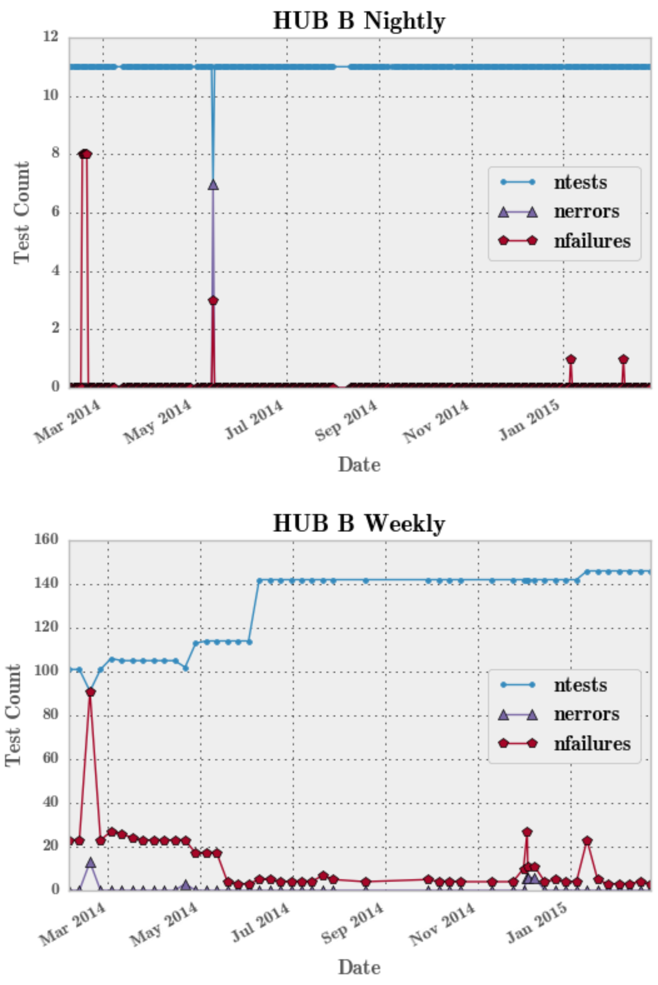


Fig. 8.2.: *Upgraded* hubs, like hub B, have a much smoother health graph, where, as time progresses and the number of tests increase, the number of failures decrease.

test suite results graph shows that with the exception of a few false positives, the number of failures holds steady at 4 tests through the end of November 2014. In the beginning of December, the software on hub B was upgraded, and HUBcheck alerted the HUBzero team to a configuration change that caused features on some web pages to be rendered incorrectly.

The gradually decreasing trend of test failures seen between February and December 2014 is characteristic of Upgraded hubs. Generally, bugs are introduced to the system at predictable times such as during system upgrades or machine reboots. By monitoring test failures through HUBcheck, the HUBzero team is able to address old problems over time and quickly stamp out new problems.

8.3 New Hubs

New hubs are the easiest of hubs to identify. Their nightly and weekly test suite results start off with nearly zero test failures and over time, new failures rarely occur. Hubs categorized as New hubs can be reclassified as Upgraded hubs after they go through a code update cycle. Newly instantiated hubs managed by the HUBzero team receive a code update on a set schedule of about every month. This strategy keeps the hubs up to date with bug fixes and contributes to the reduced number of failures seen in HUBcheck's results.

In 2014, the HUBzero team brought up five new hubs. After the initial setup was complete, each hub's weekly test suite results showed that the number of test failures was kept at a steady rate, with nearly all failures being categorized as known bugs, planned failures, or false positives.

Hub C is was one of the first hubs to be launched in 2014. Since May 2014, hub C's weekly suite results graph has consistently reported under seven failures. In June 2014, after additional tests were added to the weekly suite, the number of test failures rose very slightly due to the identification of known bugs in the HUBzero software. Through out September 2014 and November 2014, four false positives crept up, but for the most part, errors identified by HUBcheck have stayed very low since the launch of the hub.

New hubs start off with the benefit of being installed with the latest hub code and configurations. Over time, new features are introduced to the HUBzero software, bugs are fixed, and the new hubs need to be updated. Consistent updates and automated tests performed by HUBcheck help keep new hubs working properly.

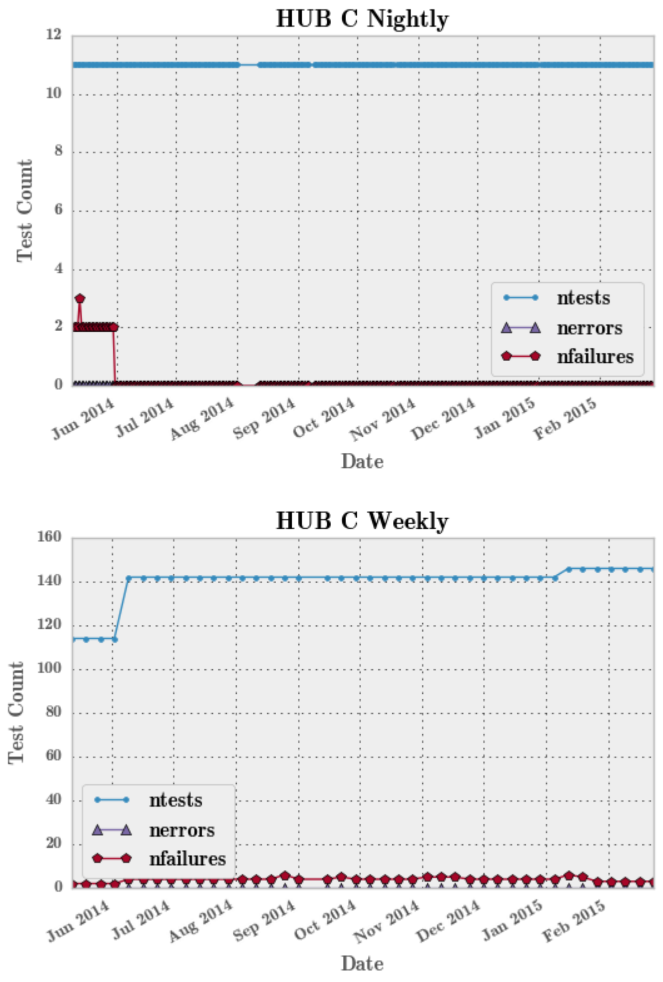


Fig. 8.3.: For *New* hubs, like hub C, once the setup has completed, the main source of test failures is known problems in the HUBzero's core software.

9. FUTURE WORK

The future of HUBcheck is full of opportunities ranging from core library improvements, to the manifestation of robust test environments, to evangelizing HUBcheck to the HUBzero Team.

9.1 Library Improvements

The HUBcheck library's shell modules are based on the assumption that tests will be performed in a tool session container over SSH, but having access to a bash shell on the local machine could open up alternative ways of running testing currently unavailable to HUBcheck. With local shell access, HUBcheck could be installed on a hub and run as a user to perform tests without needing to navigate layers of authentication, reducing false positives stemming from expired passwords and revoked SSH access. The current bash shell module template, `hubcheck.bashshell`, builds upon the `pexpect` module, a Python implementation of Tcl's Expect. The `hubcheck.bashshell` module aims to implement an interface similar to that of the `hubcheck.sshshell` module, providing the usual `send()` and `expect()` methods, but also provides an `execute()` method described in Section 6.3.4.

The organization of the HUBcheck library could also be improved. Currently, the core HUBcheck library, HUBzero specific page objects, and tests are all kept in the same repository and installed together. A better setup would be to install the core library and allow the HUBzero specific page objects to exist in a different location. This separation would also make it easier for users to build and use their own page objects while using the core HUBcheck library as a frame for developing automation scripts. The tests should be installed in a separate location, making them easy to update and add to without the need to reinstall the whole HUBcheck library.

A reorganization of the HUBcheck library should be performed in coordination with a standardization of the HUBcheck install process. HUBcheck installations should try to use system libraries whenever possible instead of shipping with its own copies of libraries to reduce the propagation of security bugs.

9.2 Test Environments

HUBcheck is one piece of the larger testing puzzle that the HUBzero Team is trying to solve. The placement of HUBcheck in the development cycle is a bit off. Right now, it runs on live hubs managed by the HUBzero Team because other available systems that are used for testing fall short in possessing the properties of a true testing environment that (1) represents a production environment, (2) is easily reproducible, and (3) is reliably available. Developers need a test environment they can quickly launch with a production compatible hub installation, update with a new feature being developed, and run HUBcheck tests. This falls in line with the goals of the HUBcheck project of lowering the barriers to adopting better testing practices earlier in the development cycle.

9.3 Adoption Within the HUBzero Team

Another missing piece to the HUBcheck project is an educational component. Using and setting up HUBcheck to run is undocumented for the most part. Future work will include evangelizing for HUBcheck, and better testing practices in general. HUBcheck is not the correct tool for all problems, but fits a unique niche within the hub environment that discouraged testing in the past and left hubs in a precarious state. The HUBcheck project helps promote one of the core phases in the software development process.

LIST OF REFERENCES

LIST OF REFERENCES

[1] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 3:1–3:30, 2012.

[2] *IBM Security AppScan*, Feb. 2013 (retrieved February 09, 2015). `http://www-03.ibm.com/software/products/us/en/appscan/`.

[3] S. Stewart, *Selenium WebDriver*, (retrieved February 09, 2015). `http://aosabook.org/en/selenium.html`.

[4] *Sahi - Web Automation and Test Tool*, (retrieved February 09, 2015). `http://sourceforge.net/projects/sahi/`.

[5] M. McLennan and R. Kennell, "Hubzero: A platform for dissemination and collaboration in computational science and engineering," *Computing in Science and Engineering*, vol. 12, no. 2, pp. 48–52, 2010.

[6] *nanohub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://nanohub.org/usage`.

[7] *nees.org Usage: Overview*, (retrieved Oct 02, 2013). `https://nees.org/usage`.

[8] *pharmahub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://pharmahub.org/usage`.

[9] *vhub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://vhub.org/usage`.

[10] *stemedhub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://stemedhub.org/usage`.

[11] *ccehub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://ccehub.org/usage`.

[12] *habricentral.org Usage: Overview*, (retrieved Oct 02, 2013). `https://habricentral.org/usage`.

[13] *molecularhub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://molecularhub.org/usage`.

[14] *purr.purdue.edu Usage: Overview*, (retrieved Oct 02, 2013). `https://purr.purdue.edu/usage`.

[15] *iemhub.org Usage: Overview*, (retrieved Oct 02, 2013). `https://iemhub.org/usage`.

[16] *c3bio.org Usage: Overview*, (retrieved Oct 02, 2013). https://c3bio.org/usage.

[17] *cleerhub.org Usage: Overview*, (retrieved Oct 02, 2013). https://cleerhub.org/usage.

[18] *drinet.hubzero.org Usage: Overview*, (retrieved Oct 02, 2013). https://drinet.hubzero.org/usage.

[19] *iashub.org Usage: Overview*, (retrieved Oct 02, 2013). https://iashub.org/usage.

[20] *diagrid.org Usage: Overview*, (retrieved Oct 02, 2013). https://diagrid.org/usage.

[21] *memshub.org Usage: Overview*, (retrieved Oct 02, 2013). https://memshub.org/usage.

[22] *geoshareproject.org Usage: Overview*, (retrieved Oct 02, 2013). https://geoshareproject.org/usage.

[23] *catalyzecare.org Usage: Overview*, (retrieved Oct 02, 2013). https://catalyzecare.org/usage.

[24] *OpenVZ Linux Containers Wiki*, (retrieved Feb 09, 2015). http://wiki.openvz.org.

[25] *Virtual Network Computing*, (retrieved Feb 09, 2015). https://en.wikipedia.org/wiki/Virtual_Network_Computing.

[26] P. Smith, T. Hacker, and C. Song, "Implementing an industrial-strength academic cyberinfrastructure at purdue university," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–7, April 2008.

[27] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, I. Foster, R. Gardner, M. Wilde, A. Blatecky, J. McGee, and R. Quick, "The open science grid," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012057, 2007.

[28] N. Wilkins-Diehr, D. Gannon, G. Klimeck, S. Oster, and S. Pamidighantam, "Teragrid science gateways and their impact on science," *Computer*, vol. 41, pp. 32–41, Nov 2008.

[29] D. Libes, *Exploring Expect: a Tcl-based toolkit for automating interactive programs.* O'Reilly Media, Inc., 1995.

[30] *Selenium WebDriver*, (retrieved February 09, 2015). http://www.seleniumhq.org/projects/webdriver/.

[31] P. Lightbody, *BrowserMob Proxy*, (retrieved February 09, 2015). http://bmp.lightbody.net/.

[32] R. Pointer and J. Forcier, *Paramiko: Python SSH module*, (retrieved February 09, 2015).

[33] *HAR 1.2 Spec*, (retrieved February 09, 2015). http://www.softwareishard.com/blog/har-12-spec/.

[34] *Representational state transfer*, (retrieved February 09, 2015). https://en.wikipedia.org/wiki/Representational_state_transfer.

[35] *Curl URL Request Library*, (retrieved April 08, 2015). http://curl.haxx.se/.

[36] D. Burns, *Python Browsermob Proxy Library*, (retrieved February 09, 2015). http://oss.theautomatedtester.co.uk/browsermob-proxy-py/.

[37] T. Schlauch, *Python client-side WebDAV Library*, (retrieved February 09, 2015). https://launchpad.net/python-webdav-lib.

[38] L. Dusseault, *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*, June 2007 (retrieved February 09, 2015). https://tools.ietf.org/html/rfc4918.

[39] M. McLennan, *The Rappture Toolkit*, (retrieved February 09, 2015). http://rappture.org/.

[40] *unittest - Unit testing framework*, (retrieved Feb 09, 2015). https://docs.python.org/2/library/unittest.html.

[41] *Nose is nicer testing for python*, (retrieved Feb 09, 2015). http://nose.readthedocs.org/en/latest/.

[42] *doctest - Test interactive Python examples*, (retrieved Feb 09, 2015). https://docs.python.org/2/library/doctest.html.

[43] *Test fixture*, (retrieved February 09, 2015). https://en.wikipedia.org/wiki/Test_fixture#Software.

[44] *xUnit*, (retrieved February 09, 2015). https://en.wikipedia.org/wiki/XUnit.

[45] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[46] A. Goucher and F. Cohen, *Create Robust Selenium Tests With PageObjects*, July 2011. http://www.pushtotest.com/create-robust-selenium-tests-with-pageobjects.

[47] *Jenkins Continuous Integration*, (retrieved Feb 09, 2015). http://jenkins-ci.org/.