taretto Documentation

RedHatQE

Feb 09, 2022

Contents:

Warning: EARLY PRE-RELEASE This version of Taretto is an early pre-release which is likely to change drastically until version 0.5.

Taretto is a collection of tooling to assist in performing functional testing of applications. The tools are designed to be used either separately or in combination with each other. For some tools, their integration is strong, others are more standalone. Currently Taretto offers the following tools:

- **Navmazing** - A tool design to build up complex navigation trees from simple steps. You define navigation *destinations*, including a way to check if you are already at the destination already, a prerequisite, and a step to take once the prerequisite is reached. Navmazing will then navigate to a destination by chaining the prerequisites together, skipping out early if it detects it is already there.
- **Widgetastic** - If you have the requirement to describe and interact with web based forms and user interfaces, Widgetastic can simplify and maximise code reuse. The system has a powerful View system to enable conditional view based on the values of widgets on the page. Widgetastic comes with support for basic HTML elements as well as the PatternFly library. More UI frameworks are planned in the future.
- **Sentaku** - This tool allows you to specify multiple methods on an object with the same name and then let the system decide which one to run based on either a context that you specify, or a predefined preferential list. This allows you to support multiple implementations, ie REST, UI, SSH, for a single object method, and have the system pick which one to use. The beauty of this approach is that your test body can be the same for each implementation and the context will dictate which implementation of hte method will be run.

In the future, Taretto is hoping to provide tooling for

- **Browser management**
- **pytest helpers**
- **Collections and Entities Modelling**

CHAPTER 1

Guides

1.1 Taretto Complete Tutorial

This guide is designed to show you how to construct and build a testing framework around Taretto. The beginning assumes you have only done minimal work on your current testing framework and are starting it from scratch. This does not mean that this is the only way to use Taretto. There are plenty of opportunities for Taretto to fit into an existing framework, you may like to browse the various guides and tutorials to see how Taretto can help you out in these cases.

Taretto, like many other systems has an optimal design pattern to follow. Whilst many of the tools can be utilized in alternative configurations and designs, it is important to know that the best integrations will be obtained by designing your framework around these fundamental principles.

You can try all the examples in this guide by downloading and running the Taretto demo application ***sheru*** as a basic web application. The details of which are described below.

1.1.1 A Basic Application

No one likes reading a ton of documentation, so we'll try to keep this short. We are going to assume we will be testing a simple poll web application, that provides a web interface, and a REST API. We'll assume that we have a database backend modeling users, polls and votes.

- **Polls** - We will have a page that shows a **list** of the *polls* that a user administrates, a **details** page for each *poll* showing the *votes* and a page to **add** a new *poll*.
- **Votes** - We will have the page to allow a *vote* to be **cast** on a particular poll. A *user* can also see all the *votes* they have cast.
- **Users** - We will have a mechanism for a *user* to **edit** their details and a superuser who can **administrate** all of the polls and votes.

This is going to form the basic simple application that we will be testing against. **Sheru** is going to provide some very basic views and widgets that we can model and use within Taretto.

1.1.2 Taretto Guiding Principles

Taretto is designed to work best in an object oriented design. Neither your application, nor your testing modeling needs to be object oriented, but this is where Taretto excels and you will be able to leverage far more of Taretto's tooling integration if you are setup in an object oriented way.

Application Object

In Taretto we start with the concept of an **Application**. The application is the basic building block of Taretto testing. It is used to hold information about the application we are testing, such as its URL or schema. It can also hold other methods that will function on the appliance, perhaps you may have some maintenance tasks that you want to perform, or clear the entire database.

This appliance object is used to hold some important pieces of information in the Taretto testing environment. It can for example hold a browser object so that you can perform Taretto UI testing, or contextual information for sentaku.

Note: This application object is not essential, but is extremely important if you want to test against multiple instances of your application at once. It is only using this application object that Taretto can be sure which Application you are targeting with your operations.

An example of this is below:

```
class Application():
    def __init__(self, url, schema="https://")
        self.url = url
        self.schema = schema
```

Entity Objects

From an organizational standpoint, testing against any kind of application usually means that you have some kind of object that the application either lets you manipulate or access. Continuing with the object oriented approach, Taretto works best if you have modeled, even very simply, objects in your application as objects in your testing framework.

An example of a basic user object is below. Notice that the application instance is required to tie a specific user instance to a particular application instance. This then means that any operations performed on the application, automatically have access to the application instance and know which application they are bound to, something which is incredibly important for things like navigating to the object in the UI, or accessing it via REST.

```
class User():
    def __init__(self, application, username, name=None):
        self.application = application
        self.username = username
        self.name = name or username
```

Widgets

In a UI there are controlling elements which allow you to interact with the page. These elements can sometimes be navigational, or they can allow you to input data, or even read data. In Taretto, these Widgets are modelled in Python, allowing you to abstract away the intricacies of XPATH or element locating. These Widgets often take either strings, or ids and are able to give the test writer an easily accessible object with simple methods to interact with them, such as *.read()*, *.fill()* and *is_displayed()*.

Taretto provides some libraries for commonly used widgets and even for some common UI frameworks, such as patternfly.

An example of a Widget definition is below:

```
account = Text(locator='//a[@title="Account"]')
```

Views

Taretto has the concept of views. These are pieces of a UI which can be inherited by other views allowing you to build up complex models of UI pages with very simple definitions. These views also give you an easy way to access the widgets on the page and perform operations with them. A view is a subclass of a widget and so by extension, it also has access to the *.read()*, *.fill()* and *is_displayed()* operations. This allows you to be able to read all of the form elements values on a single page, fill in multiple fields with a single operation by passing a dictionary, like filling in a form and checking to see if we are on a certain page.

A very basic view is shown below:

```
class BasePage(View):
    username = TextInput(id="username")
    password = TextInput(id="password")
    login_button = Text(locator='//button[@id="_eventId_submit"]')
```

Navigation

Using Taretto, we can define steps to perform navigation. The system used inside Taretto works best with an object oriented design and allows you to bind destinations, places you'd like to go to in the user interface, and link them to specific objects. This means that you can navigate to an object without the requirement of passing in any contextual information. The navigation system can also be very complex, allowing you to build incredibly dynamic models of navigation.

A simple navigation destination may look something like this:

```
@navigator.register(Application)
    class LoggedIn(IQENavigateStep):
        VIEW = BasePage
        prerequisite = NavigateToSibling('LoginScreen')

        def step(self):
            self.prerequisite_view.do_login()
```

Ignoring all the specifics right now, this navigation destination, or ND for short, defines a prerequisite navigation step of being on the **LoginScreen** and then describes the step for completing the navigation.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search