MozTrap Documentation

Release 1.4

Mozilla

March 23, 2015

Contents

1 Useful links **3**

2 Contents **5**
- 2.1 Release Notes . . . 5
- 2.2 User's Guide . . . 8
- 2.3 Installation . . . 46
- 2.4 Upgrading . . . 48
- 2.5 Development . . . 49
- 2.6 Deployment . . . 56

3 Indices and tables **59**

HTTP Routing Table **61**



MozTrap is a test case manager by Mozilla. We hope you like it.

CHAPTER 1

Useful links

- You can browse around read-only on our staging site
- Or download it from github and run it locally
- Visit our forum
- Feel free to ask questions: irc.mozilla.org #moztrap
- If you find any bugs, please enter them in Bugzilla

CHAPTER 2

Contents

2.1 Release Notes

2.1.1 Current - 1.5.4

Version 1.5.4

*release date - 8/27/14

- Upgrade django to latest security patch
- Better transaction handling for creating new Product Versions
- Some admin improvements

2.1.2 Previous

release date - 1/29/14

- **update copyright**
- **new and/or filtering switch** - Some fields that can have more than one value (like case name, or suite) could be filtered with an **or** or an **and** depending on what you want. Now there is a switch in advanced filtering to enable this.

release date - 12/20/13

- **persist pagesize settings**
- **support unmatched ‘‘when‘‘ and ‘‘then‘‘ syntax with bulk cases**
- **hover text for field on filter chicklets**
- **bug fixes** * priority value not saved with bulk cases * cloning test case keeps suite setting * author unknown when add new version of existing test case * saving from edit pages preserves filtered lists

---

Note: A special thanks to sh1r0 for these fixes and enhancements!

---

release date - 12/18/13

- **security fixes**
- **scroll to top when switching page** - When in the manage/results list.

release date - 9/9/13

- **Performance optimizations** - Went through many of the list screens and elsewhere and updated the django queries to optimize for performance. Also added several new db indexes to speed filtering and sorting.
- **Travis CI support** - Only developers will notice this, but now the unit tests are run by travis in Github to help us with determining the safety of a Pull Request.

release date - 8/22/13

- **test case priority field** - The new field of `priority` has been added to test cases. You can set the priority of a case to `1`, `2`, `3` `4` or no priority. You can filter and sort on this field as you can many other fields. See *test cases* for more info.
- **filtering by tester in results** - This allows you to see how many cases a specific tester has executed overall and for a specific run.
- **fix to edit tag dialog** - It wasn't loading the available cases for a product-specific tag due to a bug.

release date - 5/21/13

- **new run results of ''blocked'' and ''skipped''** - Blocked result is for when a test cannot be executed because it is blocked by functionality that prevents even starting the test. Skipped result is so that a `test manager` can specify that a test in a run should not be tested. This removes the test from the % complete calculation and can only be set by a *Test Manager* or *Admin*. See: *Result Statuses* for more info
- **filter lists sorted** - The list of items in the advanced filtering are now sorted for your convenience. Why didn't we do that before, you ask? Umm.. oops.
- **run progress** - The % complete for the test run in that environment now shows at the top of the page. It doesn't yet update after each result is submitted, only on page load for now. This is actually a click-able link to see the result details.

release date - 5/6/13

- **sort on results in runtests** - When you are executing a test run, you can now sort on the `results` field to help you find the tests that neither you, nor anyone else has executed yet. Or if filtering descending, it has the handy side-effect of sorting all failures to the top (since `f` comes before `p`).
- **filter by test description** - You can now filter by the description field of a test case. This is handy if you have some specific keywords, urls or filenames in the description that you need to find.

release date - 4/2/13

- **Scalability fixes around editing huge test suites**

release date - 3/28/2013

- **Upgrade to Django 1.4.5**
- **Bug fix for order of cases** - Test case order within suites was broken.
- **Bug fix for repeated cases** - It was possible, in some circumstances to have the same test case shown multiple times in a suite.

release date - 3/22/2013

- **Link to view result while running test** - If you want to share the result you just found with someone, clicking the result icon (like passed / failed) will navigate you to the result for that test. You can then share that link or add it to a bug, etc.

- **Case name sync** - It ends up that having unique case names for different versions of the case is confusing. This is especially true when you are selecting cases for a suite. The screen must show you one of the case names so it shows you the latest case name. This may not be the one you're thinking of if you're working on an earlier product. So to simplify this, any time you save a case, it will make all the version of that case the same.
- **Several bug fixes** - please see Pivotal Tracker for details.

release date - 1/22/2013

- **Fill Product Version Cases** - Added the ability to fill in case versions when they exist in one product version and not in another. This can be handy if you have created version 1.0 and 2.0 of your product in MozTrap, and have been adding new cases to 1.0 as you go. When it's time for 2.0, you want all those new cases to get moved forward. In this case, edit the 2.0 Product Version to fill cases from 1.0. See *Fill Case Versions* for more info.
- **Mass Tag / Untag Cases** - If you want to add a new tag to lots of cases, you previously had to edit each case and add it. Now, if you edit the tag in question, and select the product for the cases, you will see a list of available and included cases for that tag. This makes it possible to *merge tags*. See *Tags* for info.
- **Filter results by status** - You can now filter results cases by passed, failed or invalidated.
- **Page title shows location** - You can no see where in the product you are by the page / tab title.
- **other tweaks and bug fixes**

release date - 12/19/2012

- **Pinned Filters** - This feature allows you to **pin** a filter so that it remains constant for the session. This way, if you want to only see data for a particular *product* then you can pin the filter for it and everywhere you go, you only see data for that product. For more info, see *pinned filters*.
- **See test results from other users** - There has been an icon while running tests that indicates that another user has run it, and what that result is. And with this release, we added the comment from failed or invalid tests to the rollover text. In addition, this is now a button that will take you to the results details for that test case. See *Results of others* for more info.
- **Edit cases while running** - If you notice that a case needs updating while you are running it, there is now an *Edit this case* link in the upper right that will open a new tab to edit the contents of the case. See *running tests* for more info.
- **minor bug fixes** - New run series member sets start date to today, rather than that of the series itself. Creating a case, setting suite adds the case to the end of the suite order.

release date - 12/18/2012

- **Tag Descriptions** - You can now add descriptions to tags. The result is that when you execute tests, the description is displayed for each case that has that tag. This is a handy way to make notes that apply to a group of cases, like preconditions, links, etc. As always, Markdown syntax is suported. See *Tags* for more info.
- **Fixed refresh run bug** - The *test run refresh* to get newly added cases was broken. Now fixed.

release date - 12/10/2012

- **Display all case versions** - Formerly, when you looked at the `manage | cases` area, you would only see the latest version of each test case, unless you were filtering for a different version. This was confusing to many users, so now you see each distinct case version.
- **Delete distinct case versions** - Fixed where deleting one case version deleted all of them.
- **Create case no version default** - Many users were accidentally creating new cases for the latest version, when they meant to create it for an earlier version. Since the default for new cases is the latest version, this went un-noticed a lot. Removing the default makes it more deliberate.

release date - 12/03/2012

- **Sharable list links** - When you have filtered a list somewhere in the system, you can click the *link* icon next to the filter field to bring up the url that you can share to show that list. This link honors pagination and all filters. And it can be used in the management area as well as results and in test runs. This can be especially nice if you want to tell a tester to run a specific set of test cases in a run. See *Sharing Filters* for more info.
- **Test Run description while running tests** - We added the test run description field to the top of the page while running tests. This field supports markdown, so you can put links and other instructions to your testers in there. This can be especially helpful to add links to creating a new bug in your bugsystem of choice. (You **ARE** using Bugzilla, aren't you?) See *Run Edit Fields* for more info.
- **Filtering performance** - In some screens, the auto-complete filters were being displayed for every keystroke. Now they always wait till you're done typing before showing auto-complete options.
- **Run activation scalability** - Using some new features in Django 1.4 and a couple raw queries, we expanded support for test runs from ~700 cases to several thousand.
- **Update active test runs** - The new *refresh* button in the management area will update an active run to newly added or removed test cases. See *Refreshing a Run* for more information.
- **Case import management command** - The feature for importing cases would prevent you from importing duplicates, even if you wanted to. So added a param for that. It also accepts a directory of several files instead of just a single file.
- **Django 1.4.2 upgrade**
- **More non-ascii character fixes** - Primarily in some views and messages.
- **Split-the-work:** When you and others are executing the same test run, for the same environment, you'll see an icon on test cases where another tester has already submitted results. You can still submit your own result if you choose, but this way you don't duplicate effort, if you don't want to.
- **Test case ordering** - As you drag and drop cases in the edit Suite screen, that order will be honored when users run your tests. Same goes for suites of test runs. So, the order will be first by suite, then by case within the suite. There is also a new field in the runtests area where, if you sorted by case name, you can re-sort by order, if you like.
- **Performance fix for editing large suites** - Scalability fix as thousands of cases had been entered into the system.
- **Run Series:** See *Test Run Series* for more info on this new feature.
- **Better i18n support** - Added more support for non-ascii characters.

2.2 User's Guide

This guide provides documentation of how to use MozTrap as well as some of its underlying concepts and design decisions.

New to using MozTrap? See the *Tutorial* for an overview of some basic tasks to get you rolling.

If you find an issue with MozTrap, please enter a bug in Bugzilla



2.2.1 MozTrap Tutorial

This tutorial assumes that your instance of MozTrap has been fully installed and that you are a user with the role of *Admin*.

Note: This tutorial is a work in progress and is, as yet, incomplete.

Contents

MozTrap Tutorial, part 1

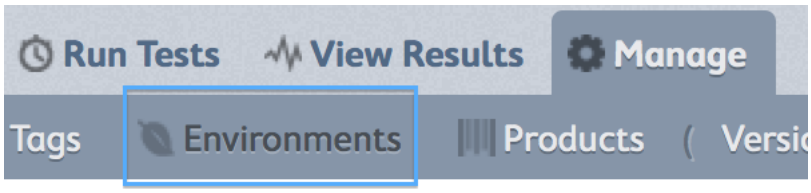
The following is a description of how to setup your new system to test your product. Part 1 will focus on setting up your Product and Environments.

Setup Environments Odds are you will be testing your product in several *environments*. These could be a collection of hardware devices, web browsers, operating systems, or even spoken languages. You want to be sure to have appropriate coverage so that you can ensure quality in the environments your product will be used.

For this we have *Environment Profiles*.

It's a good idea to familiarize yourself with the pieces that make up an environment, so consider reading *environments* before continuing.

To create a new environment profile customized to your needs, follow these steps:

1. **Navigate to `manage | environments`:**
   - 

2. Click `create a profile`.
3. Give your profile the name `Speck Envs`.
4. ---

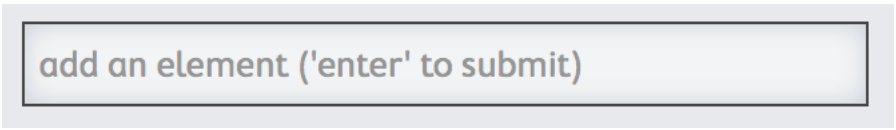
   Note: Depending on your setup, you may have several *environment elements* in your system already.

If you see a category that applies to your product already, then expand It to choose the elements that apply. If not, then click 

to create a new one.

---

5. In our case, we'll need to create everything, so click 

and type `location` and hit enter.
6. In the category, find the field that says 

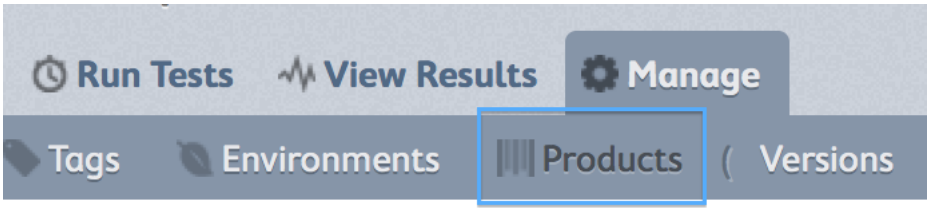
and type `laboratory` and hit "enter".
7. Add another element called `field` and hit enter.
8. Make sure to select the `location` category checkbox and all its elements.
9. Click `save profile`.

Note: This will create a matrix list of all possible combinations of the environment elements for each category you chose. In our case it's very simple (only 2). However, for other products, you may have several categories. It may be true that you don't want to test ALL combinations that were created. If that's the case, then you can winnow down the list to test.

To winnow down the list of environments to test:

1. Click the edit  icon next to your environment profile.
2. Exclude any environment by clicking the **X** next to it.
3. Click `done editing`.

Create a Product Now that you have your environment profile setup, let's create your product. We will presume your product is called **SpeckDetector**. It detects specks. Very handy.

1. **Navigate to `manage | products`:**
   - 

2. Click `create a product`.
3. Fill out the name and description.
4. Set version to 1.0. see *Product Versions* for more info on how version naming works.
5. Set the environment to the environment profile you created earlier. Or you can optionally leave it blank and add them later.
6. Click save.
7. You now have a product and version!

MozTrap Tutorial, part 2

In this section, we discuss creating test cases and organizing them into suites.

Create test Suites Test Suites are collections of test cases. A test case can belong to more than one suite, if need be.

Let's write some tests to cover two areas of the **SpeckDetector**. It should detect specks of sand and specks of pollen. And you should also be able to update your SpeckDetector's firmware.

Steps

1. Navigate to `manage | suites`.
2. Click `create a test suite`.
3. Set your product to `SpeckDetector`.
4. Set name to `Specks`.
5. Enter a description that includes Markdown syntax:

```
PRECONDITIONS
=============
* Must have some specks

LINKS
=====
* [Specks of Life](http://example.com/)
```

6. You won't have any available cases yet, so skip that and just click `save suite`.
7. Repeat these steps for a suite but name it `Firmware`.

Create test Cases Now we need to create some test cases for those suites.

Steps

1. Navigate to `manage | cases`.
2. Click `create a test case`.
3. Set your product to `SpeckDetector`.
4. Set version to `1.0`.
5. Set suite to `Specks`.
6. *ID Prefix* is optional, skip it for now.
7. Set name to `Detect a pollen speck`.
8. For `instruction` 1, enter:

```
hold detector held away from pollen
```

9. For `expected` 1, enter:

```
no detection lights
```

10. Tab to `instruction` 2, enter:

```
hold detector above a pollen speck
```

11. Tab to `expected` 2, enter:

```
detector lights up word "pollen"
```

12. Click `save test case`.

That's one down. Whew! OK, now create another test case for the `firmware` suite with steps like this:

1. Name: `update firmware`.
2. For `instruction` 1, enter:

```
navigate to firmware update screen and select "update"
```

3. For `expected` 1, enter:

```
see "a firmware update is available"
```

4. Tab to `instruction` 2, enter:

```
click "apply update"
```

5. Tab to `expected` 2, enter:

```
firmware value should say the new version
```

Great! You're done with your cases!

Moztrap Tutorial, part 3

In this section, we use the pieces you've already built to create and activate a test run that users can execute.

Create a Test Run Test Runs are made up of test suites and are specific to a version of your product. You may want to have several test runs. One could be called `smoke` and another `feature complete` and yet another `full functional tests`. Or you could break them up into larger functional areas like `front-end` and `server`.

Let's create your first **SpeckDetector** test run. It will contain all the suites you have created so far. Let's call this `feature complete`.

Steps

1. Navigate to `manage | runs`.
2. Click `create a test run`.
3. Set your product version to `SpeckDetector 1.0`.
4. Set name to `feature complete`.
5. Enter a description that includes Markdown syntax. This information will be displayed at the top of each page while running the tests:

```
LINKS
=====
* [Specks of Life](http://example.com/)
* [Bugzilla](http://bugzilla.mycompany.com)
```

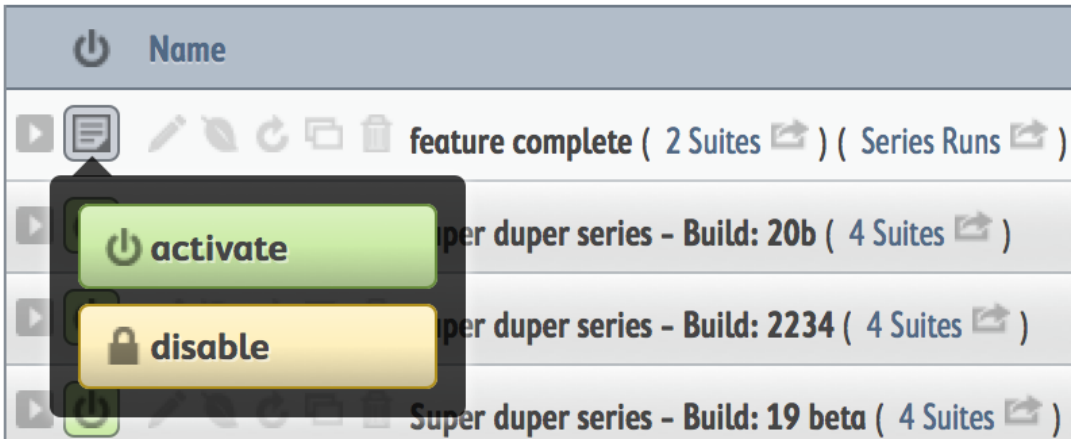
6. *series* defaults to true. We will want to run our tests against several ongoing builds of the **SpeckDetector**, so in our case we *will* create a series. Please take a moment to see what a *run series* is.
7. Leave the `start` date as today. If you want the run to expire, then set the `end` date, too.
8. Drag both suites from `available` to `included`.
9. Click `save run`.

Activate your Run Your run is just about ready. However, there's one more critical step you must take before it can be executed. You must make the run *active*.

Why not have test runs active all the time? Good question. *Look here*, Curious George.

Steps

1. Navigate to `manage | runs`.
2. Find your test run `feature complete`.
3. **Click the status icon.**



4. Click "Activate".

Isn't this exciting? You now have a test run series created and ready to go! Go tell your boss.

Moztrap Tutorial, part 4

You have now built all the parts you need to start testing your product. Allons-y! (Let's go!)

Start Testing There are a few ways to get to your test run to execute it.

Run Tests Steps

1. Navigate to `run tests`.
2. In the finder, click `SpeckDetector`.
3. Click `1.0`.
4. Click `feature complete`.

5. This is a *run series* so you will be asked to enter a `build`. Let's pretend this is your 5th feature complete build. Here type: `FC-5`
6. Set `location` to `field`.
7. Click `run tests in feature complete`.

Manage Runs Steps

1. Navigate to `manage | runs`.
2. Find the `feature complete` run.
3. Expand the arrow on the left to display the details of that run.
4. Click the green button that says `run tests in feature complete`.
5. ---

   Note: if you want to send this URL to your testers in an email, then just right-click that same button and select `copy link location`.

   ---
6. Specify your environment, as above.

"I got a URL!" Steps

1. If somebody gave you a URL to their run or run series, then click on it.
2. Specify your environment, as above.

Pass a Test Some tests pass, some fail. This is the way of the world. Let's pass this one.

1. Click the title or expansion arrow of `update firmware`.
2. Click `pass test`.
3. That was easy.

Fail a Test

1. Click the title or expansion arrow of `Detect a pollen speck`.
2. Click `fail test` next to the first step.
3. You must provide some explanation for the failure:

   ```
   We applied the cortical electrodes but were
   unable to get a neural reaction from the
   pollen speck.
   ```

4. Specifying a bug URL is optional, but it's a good idea. I'll leave that up to you.

You're done with the run! This is fantastic! If only those kids from High School could see you *now!*

Moztrap Tutorial, part 5

View Results Coming soon!

Moztrap Tutorial, part 6

Create New Version Now let's say that you shipped version 1.0 and are ready to start testing version 2.0 of the **SpeckDetector**.

Steps

1. Navigate to `Manage | Versions`.
2. Click the "create a version" button.
3. Set the product to the `SpeckDetector`.
4. Specify the version to copy Environments and Cases from. In our case, this will be `1.0`.
5. Type in the name of the new Version. In our case: `2.0`.
6. Codename is optional.
7. Click the "save productversion" button.

Result Now you will have a new product version, and a new `2.0` version of each test case. If you change the `2.0` version of a case, the `1.0` version remains unchanged. This is so that the steps in your test can evolve as your product does without changing the tests that applied to earlier versions.

Migrate Runs Test runs are specific to a version of your product. But you can easily make copies of runs from one to the other.

Steps

1. Navigate to `Manage | Runs`.
2. Find the test run you want to use in 2.0.
3. Click the clone button for that run.
4. The new run will have the name "Cloned: foo" and be in DRAFT mode.
5. Edit the newly cloned run. **Note:** It must be in DRAFT mode to change the product version field.
6. Update the name as you wish.
7. **Change the Product Version field to the new version 2.0:**

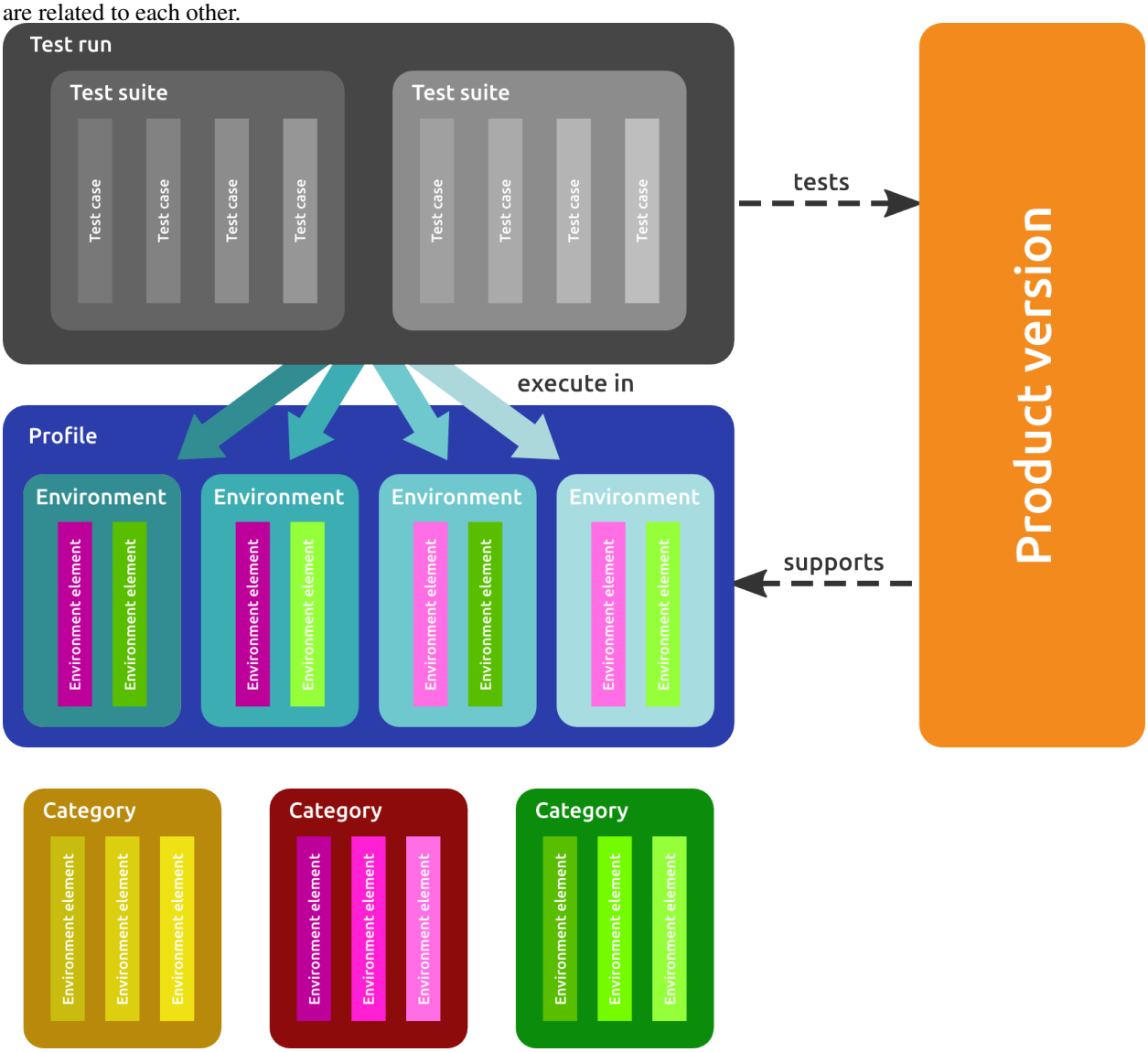


8. Save as status active, or...
9. Activate the new run with the status drop-down.

Result Now you will have a new run that applies to your new product version `2.0` that is ready to be executed.

Concepts overview

This is a rough sketch of the concepts in MozTrap to provide an overview of how the elements described in the tutorial are related to each other.



Everything revolves around the *product* under test in a specific *version*.

Test runs on the product version are executed by assigned testers. The test instructions are contained in *test cases* which are grouped in *test suites* for assignment to the runs.

The tests are run in defined *environments* whereupon the *environment profile* specifies the environments supported by the product version.

The environments themselves are composed of *environment elements* chosen from *categories*, one element per category.

2.2.2 Indices and tables

- *genindex*

- *modindex*
- *search*

2.2.3 How To Do Some Common Tasks

- Create a new Product Version for an existing Product
- Fill in test cases missing from one Product Version into another
- Migrate a test run to a new Product Version
- Get a link to a result of a test just after submitting it

Create a new Product Version for an existing Product

Situation

You have an existing Product, Version 1.0 along with test cases, suites, etc. Now you want to start testing version 2.0 of that same product. See *Product Versions* for more info on the fields in that screen.

Steps

See Tutorial: *New Version*

Fill in test cases missing from one Product Version into another

Situation

You have a 1.0 version of your product, and you created a 2.0 version as well. Then later, you added some new test cases to 1.0 and want to make sure those get included for testing in 2.0.

There are two situation variants, with different solutions:

1. You only have a couple cases you want ported to 2.0. And/or you have some cases in 1.0 that you do *not* want ported to 2.0.
2. You have *lots* of new cases in 1.0 and want them all ported to 2.0.

Steps for solution 1

1. navigate to `Manage | Cases`
2. filter as appropriate to find the cases you want to port over to 2.0
3. edit one of the cases
4. In the upper right corner of the edit page, find the drop-down beneath the `select environments` button that shows the current version 1.0 of the test case.
5. as you hover your mouse over the 1.0 version, the field will drop-down and you'll see an option that says: `+ 2.0 (add this version)`
6. Select that option

7. click `save test case`
8. edit the next test case to port and repeat the steps to add the 2.0 version.

Result

You will see new 2.0 versions of each test case you edited.

Steps for solution 2

1. navigate to `Manage | Versions`
2. edit your 2.0 (destination) version. **Note:** You can fill cases from 2.0 back to 1.0, if you like, too. Just edit the version that is your destination.
3. set the `Fill Cases From` field to the product version to fill from.
4. click `save productversion`

Result

All test cases in 1.0 now have a 2.0 version. If a 2.0 version already existed for a case, it will NOT replace it.

Migrate a test run to a new Product Version

Situation

You have an existing Product, Version 1.0 along with test cases, suites, runs, etc. You created version 2.0 of the product and want to run some of the same test runs against from 1.0 against 2.0. You will need to clone the 1.0 test runs to 2.0 and update them.

Steps

See Tutorial: *Migrate Runs*

Get a link to a result of a test just after submitting it

Situation

You are executing a test run and find a bug. You want to submit the bug, and then provide a link to the result in your bug report. Or perhaps you just want to email that link to someone.

Steps for solution 1

1. run the test run
2. mark the case with the result you found
3. **If, for example, the case was marked "Failed," then click the red button that says:**

- 

Result

You are taken to the result for that test case. You will also see any result that other users have submitted for the same case in that environment.

2.2.4 Frequently Asked Questions

1. **Why don't all of my test cases don't show up when I execute my test run?**

- Your *test cases* or *test suites* may not have been active at the time the *test run* was made active.
- When you have activated new test cases and/or suites and want a test run to reflect that, simply find the run in the management area and click the *refresh* button next to it. See *Refreshing a Run* for more info.

2. **Why don't I see the results I expect when I type in a filter?**

- When you type text into the simple search field, you'll see a drop-down list showing some possible choices. On the right of that list is the field to which that filter will be applied. If you filter for the word "Red" in the *product* field, but there is no product with the word "Red" in it, then you may see a list with no results. When you type your filter word, use the arrow keys to select the field to filter on.

3. **How can I create a test case with no steps?**

- By default, all test cases have steps, and a step has a required field of *instruction*. If you try to save the case when there is an empty instruction, it will say that you must fill out that field. To avoid this, simply click the "X" next to that step, it will be deleted, and you can save your case without steps.

4. **Please help us add more to the FAQ!**

- New FAQ items help everyone. Your contributions help!

2.2.5 Products

The core object in MozTrap is the **Product**. A Product itself is little more than a name and optional description, but almost every other object in the MozTrap data model relates to a Product either directly or indirectly.

Products have a list of *versions*; every *test run* and *test case version* applies to a particular version of the product.

Product Edit Fields

- **Name** - The name of the Product. (Firefox, Thunderbird, etc)
- **Description** - (optional) A brief description of the product.
- **Version** - Every Product must have at least one Product Version. Many Products will end up with several Product Versions. (1.0, 2.0, 2.5, etc). If this is a web project and you don't want several versions, feel free to call this whatever you like (Production, Current, etc.).
- **Environments** - This is a pre-existing collection of environments called an *Environment Profile*. You can specify this at creation time, or later. Note that the set of environments can be different for different Product Versions because the needs of your product may change over time. When you want to update the list of supported environments, you do this on the Product Version rather than the Product itself.

2.2.6 Product Versions

When a new **Product Version** is created, all test cases for that Product will get a new version to match the new **Product Version**.

For more information on how Test Cases and Product Versions relate while running tests against different builds of a Product, see the *Test Runs* section.

Product versions are automatically ordered according to their *version* number/name. The version is split into dotted segments, and the segments are ordered lexicographically (with implicit left-side zero-padding of numerals to avoid e.g. "2" ordering after "11"). So, for instance, version *1.1* is greater than version *1.0.3*, version *2.0b1* is greater than *2.0a3*, and *3.11.1* is greater than *3.2.0*.

There are some special cases to better support common version-numbering schemes. Strings alphabetically prior to "final" are considered pre-release versions (thus *2.1a*, *2.1alpha*, and *2.1b* are all prior to *2.1*, whereas *2.1g* is considered a post-release patchlevel). The strings "rc", "pre", and "preview" are considered equivalent to "c" (thus also pre-release), and the string "dev" orders before "alpha" and "beta" (so *2.1dev* is prior to *2.1a*).

Product versions can also optionally have a *code name* that does not impact their ordering.

Fill Case Versions

Test cases have a version for each Product Version. If you have multiple product versions, it is possible to have a version of a case for one product version and not for another. For example, given:

- Product Foo
  - Version 1.0
  - Version 2.0
- Case A
  - Case A, Version 1.0

You can see here that you have a version of the *Case A* for *Product Foo Version 1.0*, but not for *Version 2.0*. With a large project, you may find yourself with hundreds of cases where you created them for Version 1.0 and not for 2.0.

If you want to create those versions, you have 2 options:

1. If you only have a few, you can edit the case in question, and in the upper right of the dialog, click the version field and select *+2.0 (add this version)*
2. Edit the product version and specify the other version in the `Fill Cases From` field.

Product Version Create Fields

- **Product** - The Product that this is a version of.
- **Copy Environments and Cases From** - (optional) Environments apply to each product version. Each version can have a unique set of environments. But commonly, they are very close, and the set of environments evolves over time. This field allows you to choose which existing product version to copy the environments from. You can then add or remove from the list of environments for this version.

Fields in both Create and Edit

- **Version** - The name of the new version. See *product versions* for more info on how order of versions works.

- **Codename** - (optional) This can be any text and is only used as a reference in the summary list of versions when there is another name for a version. For instance, for Mac OS 10.7, the Codename is *Lion*.

Product Version Edit Fields

- **Fill Cases From** - (optional) The product version to copy cases from if they don't exist for this product version yet. See *Fill Case Versions*.

2.2.7 Test Cases, Suites and Tags

Test Cases

A **Test Case** is a named set of steps for testing a single feature or characteristic of the system under test. Test cases are associated with a *product*, and can have one version per *product version*. They can be organized via *suites* and/or *tags*, and can have file *attachments*. Preconditions, assumptions, and other preliminary information can be provided in the case's *description*. A test case can have any number of steps; each step has an *instruction* and an *expected result*.

Case Edit Fields

- **Product** - The product that owns this test case.
- **Version** - The product version of this test case.
- **And Later Versions** - Create a test case version for the specified Product Version as well as a case version for each later Product Version. (e.g.: if Product Versions 3, 4 and 5 exist for this Product, and you have specified Product Version 4, this case will be created for versions 4 and 5)
- **Suite** - (optional) The existing suite to which you want this case to belong. You can also add cases to suites later.
- **ID Prefix** - (optional) A string that will be displayed as part of the case ID. This can be a component name, or any string that is pertinent. This is also supported when filtering by ID. You can filter by the prefix only, by the ID, or by the prefix-ID combination.
- **Name** - The summary name for the case.
- **Description** - Any description, pre-conditions, links or notes to associate with the case. This field is displayed while running the test. Markdown syntax is supported.
- **Priority** - The priority of the test case. Set a case as `priority 1` to indicate it is the highest priority. This can be anywhere from no priority, or from 1 through 4. This is the same across all versions of the case. You can filter and sort by this field when running or managing cases. You can also filter by this when selecting which cases to include in a suite.
- **Add Tags** - Enter tags to apply to this case. Hit enter after each tag to see the tag chicklet displayed. Auto-completes for existing tags. During test execution, cases that have tags will show the tag descriptions with with each case.
- **Add Attachment** - You can attach files to cases that may help running the test. (e.g: images, audio, video, etc.)
- **Instruction / Expected** - The test instruction and corresponding expected result. You can choose to put all instructions / expectations in one step, or break them down to individual steps. When running the test, you will have the option to fail on specific steps, so you may find this a better approach. Markdown syntax is supported.
- **Save** - You can choose to save the case as draft or active. Only active cases can be run in a test run.

Test Suites

A **Test Suite** is a named collection of test cases that can be included in a *test run*.

Suite Edit Fields

- **Product** - The product that owns this test case.
- **Name** - The name of the suite.
- **Description** - Any description for the suite.
- **Available Cases** - Test Cases that have the same Product you selected for this suite. This list is filterable.
- **Included Cases** - Test Cases that are included in the Suite. This list is not filtered.

Tags

A **Tag** can be associated with one or more *test cases* as a way to organize and filter them on any number of axes.

By default, tags are *product*-specific; global tags can also be created and managed via the tag management UI.

Merging Tags

The edit screen for tags is a great way to merge two tags into one. For example, if you wanted to merge TagA and TagB all into TagB, then simply:

- Edit TagB
- In the list of available cases, filter on TagA
- Select all the available cases and click the green add button
- Save TagB
- Delete TagA

Tag Edit Fields

- **Name** - The name of the tag.
- **Product** - (optional) Tags can be specific to a Product, or they can be global. If a tag is Product specific, then cases for other products can't use it. This is useful if you want to separate tags for different products.
- **Description** - (optional) This description will be displayed during test execution before the test case description and steps. This is useful to provide some *setup* or *precondition* code that doesn't have to be repeated for a group of cases. Supports Markdown syntax.
- **Available Cases** - Test Cases that have the same Product you selected for this tag. This list is filterable.
- **Included Cases** - Test Cases that have this tag applied. This list is not filtered.

Attachments

A *test case* can have any number of file attachments: these will be made available for download by testers when the test case is executed.

2.2.8 Runs and Results

Test Runs

A **Test Run** consists of a set of *test case* versions that can be assigned to a tester for execution (or that a tester can assign to themselves and execute) in a particular *environment* or set of environments.

A test run is for a specific *product* version. It has its own *name*, *status*, *start date*, and *end date*, as well as a list of included *test suites*. A test run must be switched to *active* status before it can be executed by testers.

A Test Run applies to a Product and a Product Version. Usually, a product has had several iterations (or builds) prior to the release of a final Version. Therefore, a Test Run is a single execution pass over a specific iteration of that Product Version. And your product will likely have more than one iteration prior to release of that version. For this purpose, you may want to make a Test Run that is a *Series*.

The test case steps executed in test runs may be different for each Product Version, as the Product itself evolves. See *Test Cases* for more info on how test case versions relate to Product Versions. Draft test runs cannot be executed yet. This is a good state if you're still working on it and aren't ready for people to see it. It won't show up in the list of test runs for your product in the `Run Tests` section.

An active test run can be disabled or made draft, which halts all execution of tests in that run until it is made active again.

Test Run Series

A **Test Run** can be marked as a series of runs, by checking the "Is Series" box. You define the run just as you would any other run by specifying the *product* version and suites. The difference is that, this **Series** run is now just a template used for each build of your product to be tested.

When you execute a run that is a series, you will be prompted for your environment options, like always. But you will also be asked for a build id. If there is already a member of this series that has that build id, then you will begin testing it. If, however, no one has run this series on that build yet, then a new member of the series will be created and you will start testing it. The name of this new member of the series will contain the build id you specified. For instance, with a run series called "Smoketest," specifying a build of "Alpha1" will result in a new member of the series named "Smoketest - Build: Alpha1" with distinct results from any other member in the series.

When viewing the list of runs in the manage or results lists, you can then filter to see only runs that belong to a specific series.

Cloning Test Runs

If you have a Test Run that you would like to apply to a different Product Version, you must clone the existing Test Run, then edit the new clone while it is still in draft mode. Once your changes are made, you can activate the new run to use it.

Sharing links to Runs

Often you might create a run or run series and want to send a link to your testers asking them to execute it in their own testing environment. This also works great for a run *series*. To get this link, expand the details for your run in the manage runs area. You'll see a big green button saying **run tests in <yourrunname>**. Just right-click and copy that url location to share.

Run Edit Fields

- **Product Version** - The product version of this test run. Runs are specific to a version of a product, not just the product in general.
- **Name** - The summary name for the run. When testing a product that has build numbers, you may choose to include the build number in the name to distinguish it from other runs against the same version of the product. Dates in the name are another good way to distinguish runs from one another.
- **Series** - (optional) Whether or not this run is a *series* of runs. Default to True.
- **Description** - (optional) Any description for the run. This description is displayed in the management details area as well as at the top of the page while executing a run. Description text supports Markdown syntax which could include links to things like entering a new bug in one or more areas or extra info for your testers.
- **Start** - The first date that the run can be executed
- **End** - The date the run expires. A run cannot be executed after its end date.
- **Available Suites** - All the suites that apply to the specified Product Version. This field is filterable.
- **Selected Suites** - The suites from which to gather test cases for this run. When the run is activated, only suites and cases that were active at that time will be included in the run. This field is not filterable.

Refreshing a Run

When a test run is made active, it will take a snapshot of active suites and cases at that time. If cases and suites are added, removed or had their active status changed since the run was made active, the run won't appear changed to testers. This is because once a test run is activated, it is considered a *unit of work* that you may not want to alter while testers are executing the run.

If, however, you want to refresh the run with the new list of active cases and suites, then you can click the *refresh* button in the management area next to your run. This won't affect existing results unless you have removed a case from one of the run's suites.

Test Results

A **Test Result** stores the results of a single execution of one *test case* from a *test run*, in a particular *environment*, by a particular *tester*.

A result has a *status*, which can be any of **assigned** (the test case/environment is assigned to this tester, but hasn't been run yet), **started** (the tester has started executing the test, but hasn't yet reported the result), **passed**, **failed**, or **invalidated** (the test case steps were incorrect, did not apply, or the tester couldn't understand them).

The result also tracks the duration of execution (datetime *started* and *completed*), as well as an optional *comment* from the tester.

A passed/failed/invalidated result can also be recorded for each individual step in the test case, allowing the tester to specify precisely which step(s) failed or were invalid. A failed step can have a *bug URL* associated with it.

2.2.9 Environments

MozTrap allows fine-grained and flexible specification of the environment(s) in which each test should be run.

An **Environment** is a collection of *environment elements* that together define the conditions for a single run of a test. For instance, an environment for testing a web application might consist of a browser, an operating system, and a language; so one environment might be **Firefox 10, OS X, English**, and another **Internet Explorer 9, Windows 7, Spanish**. An **Environment Element** is a single element of a test environment, e.g. **Windows 7** or **Firefox 10**.

An **Environment Category** is a category containing several (generally mutually exclusive) elements. For instance, the **Operating System** category might contain the elements **OS X 10.5**, **OS X 10.6**, **Windows Vista**, and **Windows 7**. An **Environment Profile** is a collection of *Environments* that specifies the supported environments for testing a product or type of product. For instance, a **Web Applications** environment profile might contain a set of environments where each one specifies a particular combination of web browser, operating system, and language.

Environment profiles can be named and maintained independently of any specific product; these generic profiles can then be used as the initial profile for a new product. For instance, the generic **Web Applications** profile described above could be used as the initial profile for a new web application product.

Product versions, *runs*, and *test cases* all have their own environment profile; that is, the set of environments relevant for testing that particular product version, test run, or test case. These profiles are *inherited*.

Environment Edit Fields

- **Name** - The name of the *Environment Profile*. This name is what you'll see when selecting environments for a *product version*.
- **Table**
  - **Name** - The name of each *environment category*. Select the environment categories you want to include in your profile. You can create new categories as you need them (see **Add a Category** below)
  - **Elements** - The *environment elements* that exist in this category. You can select **all** elements from a category, or specific ones. You can also create new ones, as you need.
  - **Add a Category** - Click this bar to add a new *environment category*. Just type the new category name in the field and hit enter. You can then add elements to it.
- **save profile** - Clicking this will auto-generate all combinations of the categories and elements you chose above. You will then be taken to a screen where you can pare the list of environments down to only the ones you truly want to have included in the profile. See **Auto-generation** below for more info.

Auto-generation

Given a set of *environment categories* (or subsets of the *elements* from each *category*) MozTrap can auto-generate an environment profile containing every possible combination of one element from each category.

For instance, given the *elements* **Firefox** and **Opera** in the *category* **Browser** and the elements **Windows** and **OS X** in the category **Operating System**, the auto-generated profile would contain the *Environments* **Firefox, Windows**; **Firefox, OS X**; **Opera**, **Windows**; and **Opera, OS X**.

Inheritance

At the highest level, a product version's environment profile describes the full set of environments that the product version supports and should be tested in.

A test run or test case version by default inherits the full environment profile of its product version, but its profile can be narrowed from the product version's profile. For instance, if a particular test case version only applies to the Windows port of the product, all non-Windows environments could be eliminated from that test case's environment profile. Similarly, a test run could be designated as Esperanto-only, and all non-Esperanto environments would be removed from its profile (ok, that's not very likely).

The environment profile of a test case or test run is limited to a subset of the parent product version's profile - it doesn't make sense to write a test case or execute a test run for a product version on environments the product version itself does not support.

When a test case is included in a test run, the resulting "executable case" gets its own environment profile: the intersection of the environment profiles of the test run and the test case. So, for example, if the above Windows-only test case were included in an Esperanto-only test run, that case, as executed in that run, would get an even smaller environment profile containing only Windows Esperanto environments.

Thus, the inheritance tree for environment profiles looks something like a diamond:

```
product-version
  /        \
 run     case-version
  \        /
executable-case-version
```

Cascades Whenever an environment is removed from an object's profile, that removal cascades down to all children of that object. So removing an environment from a product version's profile also automatically removes it from all test runs and test cases associated with that product version.

Adding an environment only cascades in certain situations. Adding an environment to a product version's profile cascades to test runs only if they are still in Draft state; once they are activated, their environment profile can no longer be added to.

Additions to a product version's environment profile cascade only to those test cases whose environment profile is still identical to the product version's environment profile (i.e. test cases that apply to all environments the product supports). Once a test case has been narrowed to a subset of the product version's full environment profile, additions to the product version's profile will have to be manually added to the case's profile if the new environment applies to that case.

Test results, once recorded, are never deleted, even if their corresponding environment is removed from their product version or run's environment profile.

Select Environments

This page allows you to narrow the list of environments for a given object. This can be a *product version*, *test run*, *test suite*, or *test case*. See **Inheritance** and **Cascades** above for a detailed explanation. In this dialog, you can uncheck any environments that you do not want to apply the version/run/suite/case in question. You can also add environments back in that may have been previously removed. Just check or uncheck items to include / exclude them.

2.2.10 Teams

Any *product*, *product version*, or *test run* can optionally have a **Team**, which is just a set of users. Teams are not named or managed as an independent entity; they are simply a set of users associated with a given product, version, or run.

Teams are inherited by default; any product version without its own team explicitly set will inherit its product's team, and any test run without a team set will inherit its product version's team. Unlike *environment inheritance*, there is no subset requirement - a test run can be explicitly assigned any team, even if some members of that team are not part of the product version or product's team.

When a test run is activated, all team members for that test run will automatically be assigned all test cases in that run.

2.2.11 Roles and Permissions

Default roles

Four default roles are created when you run `python manage.py create_default_roles`: *Tester*, *Test Creator*, *Test Manager*, and *Admin*. These roles can be fully customized, and new ones can be created (currently only via the Django admin at `/admin/`).

The default roles have the following permissions:

Tester

- *execute*

Test Creator

All *Tester* permissions, plus:

- *create_cases*
- *manage_suite_cases*

Test Manager

All *Tester* and *Test Creator* permissions, plus:

- *manage_cases*
- *manage_suites*
- *manage_tags*
- *manage_runs*
- *review_results*
- *manage_environments*

Admin

All *Tester*, *Test Creator* and *Test Manager* permissions, plus:

- *manage_products*
- *manage_user*

When setting up MozTrap, running the command `python manage.py create_default_roles` will ask to create an admin user. This special first admin has all these privileges:

- **Admin role in the MozTrap UI**: This gives the user the ability to visit the `Manage | Users` area of the product. This user can edit other users to:
  - assign roles
  - create api keys
  - delete or deactivate

- **Staff Status**: This gives the user access to the `/admin/` url. This is a special *behind the scenes* access to the data in MozTrap. It is also where items that were deleted can be *undeleted*.
- **Superuser Status**: A user that has this status will always have admin privileges in the MozTrap UI, even if their role is changed to something other than `Admin`.

So you can see that this first admin user is special, and also the gateway to providing access for all other users to be admins.

Permissions

execute

Can run tests and report the results.

create_cases

Can create new test cases and edit them (but not edit test cases created by others). Allows tagging of these test cases with existing tags, but not creation of new tags.

manage_suite_cases

Can add and remove test cases from suites.

manage_cases

Can add, edit, and delete test cases and test case versions.

manage_suites

Can add, edit, and delete test suites.

manage_tags

Can add, edit, and delete tags.

manage_runs

Can add, edit, and delete test runs.

review_results

Can review submitted test results and mark them reviewed.

manage_environments

Can create, edit, and delete environment profiles, categories, elements, and environments.

manage_products

Can create, edit, and delete products and product versions.

manage_user

Can create, edit, and delete users.

2.2.12 REST API

These are the REST endpoints available to MozTrap. These are build using the TastyPie package, so please also refer to the TastyPie documentation for more info.

General

The general format for all rest endpoints is:

GET /api/v1/<object_type>/
Return a list of objects

limit (optional) Defaults to 20 items, but can be set higher or lower. 0 will return all records, but may run afoul of **Example request**:

```
GET /api/v1/product/?format=json&limit=50
```

GET /api/v1/<object_type>/<id>/
Return a single object

POST /api/v1/<object_type>/
Create one or more items.

- **requires** *API key*
- **requires** username

If sending the fields as data, the data must be sent as json, with Content-Type application/json in the headers.

PUT /api/v1/<object_type>/<id>
Update one item.

- **requires** *API key*
- **requires** username

DELETE /api/v1/<object_type>/<id>
Delete one item.

- **requires** *API key*
- **requires** username

Note:

- POST does not replace the whole list of items, it only creates new ones
- DELETE on a list is not supported
- PUT to a list is not supported

- commands that make changes may need to be sent to https, not http.

Query Parameters

- See each individual *Object Types* for the params it supports.
- See TastyPie Filtering for more info on query parameters.

Some fields are universal to all requests and *Object Types*:

- **format (required) The API always requires a value of `json` for** this field.

Note: The underscores in query param fields (like `case__suites`) are **DOUBLE** underscores.

Supported Object Types

Product API

Product
`GET /api/v1/product`

Filtering

name The name of the product to filter on.

```
GET /api/v1/product/?format=json&name=Firefox
```

`GET /api/v1/product/<id>`

`POST /api/v1/product`

Required Fields

name A string Product name.

productversions A list of at least one Product Version.

Optional Fields

description A string description.

`DELETE /api/v1/product/<id>`

Note: Deleting a Product will delete all of it's child objects.

`PUT /api/v1/product/<id>`

Note: ProductVersions are displayed in the GET results. They may be added to or changed by a POST request, but a POST to Product will not delete any ProductVersion.

Product Version
GET /api/v1/productversion

Filtering

version The ProductVersion `name` to filter on. For example, if the Product and Version are `Firefox 10` then the `version` would be `10`.

product The Product `id` to filter on.

product__name The Product `name` to filter on.

Example request:

```
GET /api/v1/productversion/?format=json&version=10
GET /api/v1/productversion/?format=json&product__name=Firefox
```

GET /api/v1/productversion/<id>

POST /api/v1/productversion

Required Fields

version A string ProductVersion name.

product A resource uri of the parent Product.

Optional Fields

codename A string codename.

DELETE /api/v1/productversion/<id>

PUT /api/v1/productversion/<id>

Note: The Product of an existing ProductVersion may not be changed.

Test Cases and Suites API

For additional information, please consult https://moztrap.readthedocs.org/en/latest/userguide/model/library.html

Case
GET /api/v1/case

Filtering

product The Product `id` to filter on.

product__name The Product `name` to filter on.

suite The Suite `id` to filter on.

suite__name The Suite `name` to filter on.

GET /api/v1/case/<id>

Note: Suites are displayed in the GET results, for informational purposes, but may not be changed.

POST /api/v1/case

Required Fields

product A resource uri to a Product.

Optional Fields

idprefix A string that will be displayed as part of the case ID.

DELETE /api/v1/case/<id>

PUT /api/v1/case/<id>

Note: The product of an existing case may not be changed.

Case Version
GET /api/v1/caseversion

Filtering

productversion The ProductVersion `id` to filter on.

productversion__version The ProductVersion `name` to filter on. For example, if the Product and Version are `Firefox 10` then the `productversion__version` would be `10`.

productversion__product__name The Product `name` to filter on.

case__suites The Suite `id` to filter on.

case__suites__name The Suite `name` to filter on.

tags__name The tag `name` to filter on.

Example request:

```
GET /api/v1/caseversion/?format=json&productversion__version=10&case__suites__name=Sweet%20Suite
GET /api/v1/caseversion/?format=json&productversion__product__name=Firefox
```

GET /api/v1/caseversion/<id>

Note: Environments, Tags, Suites and CaseSteps are displayed in the GET results for informational purposes, but may not be changed.

POST /api/v1/caseversion

Required Fields

case A resource uri to the parent Case

productversion A resource uri to a ProductVersion

Optional Fields

name A string name

description A string description

status `active`, `draft`, or `disabled`

Note: The parent Case's Product must match the ProductVersion's Product.

`DELETE /api/v1/caseversion/<id>`

`PUT /api/v1/caseversion/<id>`

Note: The `productversion` and `case` fields are not required, and may not be changed.

CaseSteps
`GET /api/v1/casestep`

Filtering

caseversion The `id` of the parent caseversion

caseversion__name The `name` of the parent caseversion

`POST /api/v1/casestep`

Required Fields

caseversion A resource uri to a CaseVersion

instruction A string describing what actions to take

number An integer used to order steps

Optional Fields

expected The expected result following the instruction

`DELETE /api/v1/casestep/<id>`

`PUT /api/v1/casestep/<id>`

Note: The CaseVersion of an existing CaseStep may not be changed.

Suites
`GET /api/v1/suite`

Filtering

name The `name` of the suite

product The `id` of the product for this suite

product__name The `name` of the product

Example request:

```
GET /api/v1/suite/?format=json
```

POST /api/v1/suite

Required Fields

product A resource uri to a Product

Optional Fields

name A string name

description A string description

status `active`, `draft`, or `disabled`

DELETE /api/v1/suite/<id>

PUT /api/v1/suite/<id>

Note: The Product of an existing Suite may not be changed.

SuiteCase
GET /api/v1/suitecase
GET /api/v1/suitecase/<id>

POST /api/v1/suitecase

Required Fields

case A resource uri to a case

suite A resource uri to a suite

order An integer used to sort the cases within the suite.

Note: The Case's Product must match the Suite's Product.

DELETE /api/v1/suitecase/<id>

PUT /api/v1/suitecase/<id>

Note: Only the order may be changed for an existing SuiteCase.

Test Runs API

Test Run The Test Run (and related) API has some special handling that differs from the standard APIs. This is because there is complex logic for submitting results, and new runs with results can be submitted.

Please consider MozTrap Connect as a way to submit results for tests. You can also check out MozTrap Connect on github.

GET /api/v1/run

POST /api/v1/run

Productversion (optional) The ProductVersion ID to filter on.

Productversion__version (optional) The ProductVersion `name` to filter on. For example, if the Product and Version are `Firefox 10` then the `productversion__version` would be `10`.

Productversion__product__name (optional) The Product `name` to filter on.

Status (optional) The status of the run. One of `active` or `draft`.

Example request:

```
GET /api/v1/run/?format=json&productversion__version=10&case__suites__name=Sweet%20Suite
```

Run Case Versions
GET /api/v1/runcaseversion

Filtering

run The `id` of the run

run__name The `name` of the run

caseversion The `id` of the caseversion

caseversion__name The `name` of the caseversion

```
GET /api/v1/product/?format=json&run__name=runfoo
```

Results
PATCH /api/v1/result

Example request: This endpoint is write only. The submitted result objects should be formed like this:

```
{
    "objects": [
        {
            "case": "1",
            "environment": "23",
            "run_id": "1",
            "status": "passed"
        },
        {
            "case": "14",
            "comment": "why u no make sense??",
            "environment": "23",
            "run_id": "1",
            "status": "invalidated"
        },
        {
            "bug": "http://www.deathvalleydogs.com",
            "case": "326",
            "comment": "why u no pass?",
            "environment": "23",
            "run_id": "1",
            "status": "failed",
```

```
                "stepnumber": 1
            }
        ]
    }
```

Environment API

Environments do not behave in quite the same way in the API as they do in the Web UI. In the API, create Categories and their child Elements first, then create a Profile for which you can create Environments whose elements must each belong to a separate profile.

Profile
GET /api/v1/profile

Filtering

name The `name` of the Profile to filter on.

GET /api/v1/profile/<id>

POST /api/v1/profile

Required Fields

name A string Profile name.

DELETE /api/v1/profile/<id>

PUT /api/v1/profile/<id>

Category
GET /api/v1/category

Filtering

name The `name` of the Category to filter on.

GET /api/v1/category/<id>

POST /api/v1/category

Required Fields

name A string Category name.

DELETE /api/v1/category/<id>

PUT /api/v1/category/<id>

Element
GET /api/v1/element/

Filtering

name The `name` of the Element to filter on.

category The `id` of the Category to filter on.

category__name The `name` of the Category to filter on.

`GET /api/v1/element/<id>`

`POST /api/v1/element`

Required Fields

name A string Element name.

category A resource uri to the parent Category.

`DELETE /api/v1/element/<id>`

`PUT /api/v1/element/<id>`

Note: The Category of an existing Element may not be changed.

Environment
`GET /api/v1/environment`

Filtering

elements (optional) The Element ID to filter on.

Example request:

```
GET /api/v1/environment/?format=json&elements=5
```

`GET /api/v1/environment/<id>`

`POST /api/v1/environment`

Required Fields

profile A resource uri to the parent Profile.

elements A list of element resource uri's.

Note: Each element must be from a separate category.

`DELETE /api/v1/environment/<id>`

`PUT /api/v1/environment/<id>`

`PATCH /api/v1/environment`

The *PATCH* command is being overloaded to provide combinatorics services to create *environments* out of *elements* contained by *categories*.

To create environments for all of the combinations of elements in the listed categories:

```
data={
    u'profile': u'/api/v1/profile/1',
    u'categories': [u'/api/v1/category/1', ...]
}
```

You may also do combinatorics with partial sets of elements from the categories by using dictionaries with 'include' and 'exclude' keys.

```
data={
    u'profile': u'/api/v1/profile/1',
    u'categories': [
        {
            u'category': u'/api/v1/category/1',
            u'exclude': [u'/api/v1/element/1']
        },
        {
            u'category': u'/api/v1/category/2',
            u'include': [
                u'/api/v1/element/4',
                u'/api/v1/element/5'
            ]
        },
        {
            u'category': u'/api/v1/category/3'
        }
    ]
}
```

Note: The included or excluded elements must be members of the category they accompany. If both include and exclude are sent with the same category, exclude will be performed.

Tags API

Tag

GET /api/v1/tag

Filtering

name The Tag `name` to filter on.

product The Product `id` to filter on.

product__name The Product `name` to filter on.

Example request:

```
GET /api/v1/tag/?format=json
```

GET /api/v1/tag/<id>

POST /api/v1/tag

Required Fields

name A string name for the Tag.

product A resource uri to a Product.

Optional Fields

description A string description for the Tag.

DELETE /api/v1/tag/<id>

PUT /api/v1/tag/<id>

Note: The Tag's Product may not be changed unless the tag is not in use, the product is being set to None, or the product matches the existing cases."

API Keys

API Keys are generated on the `Manage | Users` page for a user. Only an *Admin* can create an API Key for a user.

The API key is passed on the query string for an API like this:

```
``POST /api/v1/product?username=camd&api_key=abc123``
```

2.2.13 Filtering

Quick Filtering

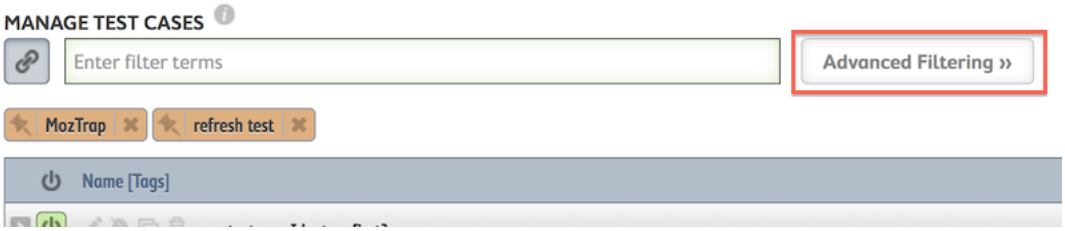


You can type into this field if you generally know the text of what you want to filter on. Auto-complete will give you several choices. Take care to select the choice with the correct `field`. For instance

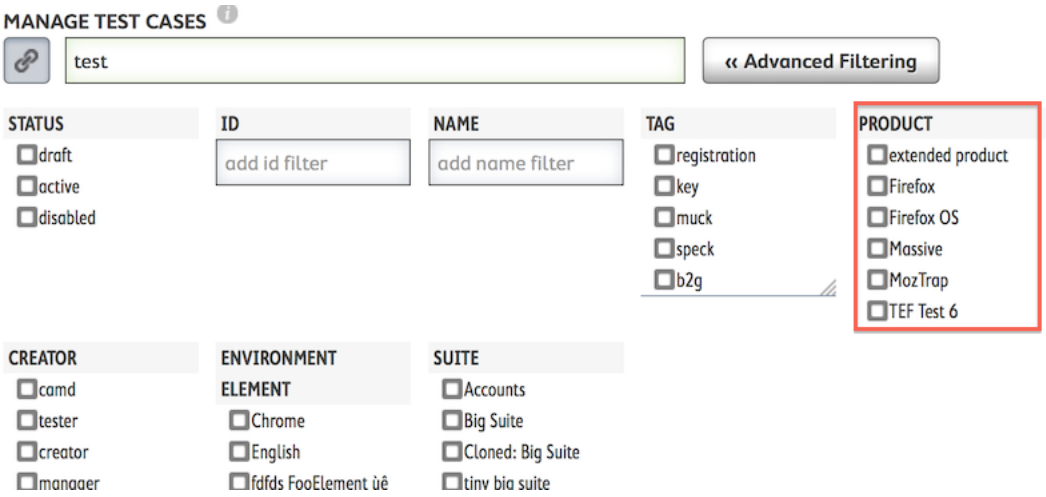


Here you can see that filtering can be done on any of these specific fields. If you filter for the word `test` in the wrong field, you may not get the results you were hoping for.

Advanced Filtering



Clicking the Advanced Filtering button will expand a list of supported filter fields for the current screen.



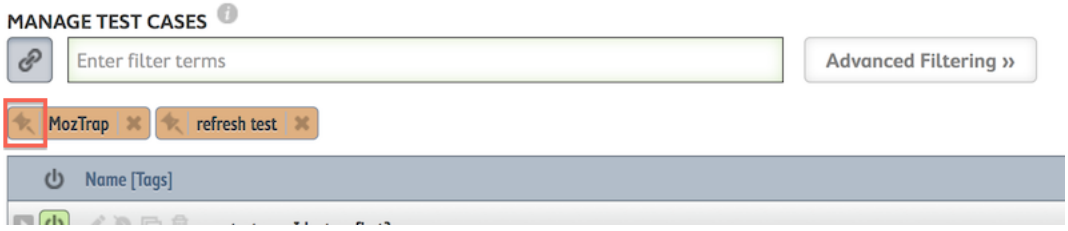
This can help be more specific in your filtering and can also help if, for instance, you don't know the exact spelling of the `product` you would like to filter on.

Sharing Filters



Click on the link button to open a drop-down that has a link you can share that includes all the filters to your current list and page.

Pinning Filters



When you have selected a filter, it may be one that you would like to **stick** wherever you go in MozTrap. That's what **pinned filters** are for. Pinning a filter will persist that choice.

Perhaps the most useful fields to pin would be the *product* field and the *product version* fields. Pinning these fields means you would only see the information that pertains to them no matter where you go in the product.

Note: Some screens may not show all of your pinned filters. For example, if you pin a **product** and **product version** in the `manage | cases` screen, you will see both pinned filters show in orange. However, if you then navigate to the `manage | suites` screen, you will notice that you only see the pinned **product** filter. This is because suites are not specific to any **product version** and therefore don't have a filter for it. Suites only have a **product** filter, so that is what you see.

2.2.14 Working with Lists

Much of the navigation in MozTrap is done with lists. When managing things like *Test Cases*, it is possible to have a very long list of items.

Filters

You can use filters to narrow down the size of the list you're currently viewing. You can do simple filtering by either clicking certain fields in the list (like tags or Product Version), or by typing them in the filter field.

Another option is to click the "Advanced Filtering" button to show your filter options. Simply click the item value you would like to use for filtering. When it has a check mark next to it, the filter is enabled.

Note: If you have two filters for items of the same type (such as two Tag filters) then the filters are treated as an OR between them, rather than an AND. For instance, if you filter on tags "One" and "Two" the list will reflect items that have EITHER "One" or "Two", not just ones that have both.

Controls

Details



Notice there is a triangle on the very left of every list item. Click this triangle to expand and see details about that list item.

Status



Many lists have items that can be in `active`, `draft`, or `disabled` state. Clicking this icon will give you a drop-down to change the items state.

- **active** - This item can be used for running tests. For example, `active` cases and suites will be included in test run execution. Active runs can be executed.
- **draft** - These items are considered *still under development* and won't be used in runs. `draft` cases can still be included in suites, and `draft` suites can still be included in runs, but they will not show up when actually executing a run.
- **disabled** - These are items that are no longer intended for use. Similar to draft mode in that they can't be used for test execution.

Edit



Navigates to an edit page where you can change the values of the item. For some items, the status may make a difference. For instance, when editing an active *test run*, you will not be able to change which suites it contains. You must change it to draft mode first.

Environment



Navigates to a page to manage which environments pertain to this item.

Clone



Make a copy of the item. For *cases* this is a good way to make a derivative version of an existing case.

Delete



Delete the item. MozTrap always uses **soft deletes**. So if you delete something and change your mind, an administrator can un-delete the item in the `/admin` panel.

Test Run Controls

Refresh Cases



Refreshes the test run cases to remove or add cases based on any changes in status made to suites or cases (including additions of new cases to suites) since this run was made active.

See *Refreshing a Run* for details.

Test Suite Controls

Add Case



Navigates to the `Create a new Test Case` page with this suite field pre-populated.

See *Test Cases* for more details.

2.2.15 Running Tests

To execute a *Test Run* select the **Run Tests** tab at the top of MozTrap. From here, select the *Product*, *Product Version* and *Run*. Then you will be prompted to enter the *environment* you are using to test.

Result Statuses

- **passed** - All steps of the test matched the expected result.
- **failed** - One or more steps of the test did not match the expected result.
- **invalid - The steps of the test were either incorrect, or unclear to the** extent that it was not possible to determine if it passed or failed
- **skipped - A test manager decided this test should not be run and should** not count against the % complete for any environment. Marking a test *skipped* marks that test skipped for ALL environments the case applies to. This is true, even if the case had been marked passed or otherwise in a different environment. Likewise, restarting a `skipped` case will restart it for all environments.
- **blocked - The test could not be run because the user was blocked from** beginning the test. For example, if the steps are to complete a purchase of something in their shopping cart, but the user can't even add items to the shopping cart, then the case could be considered blocked.

Marking a result

Expand a case to see buttons to mark the test *passed, failed, skipped, blocked or invalid.* You can fail any specific step of a test case. Marking a case invalid means that the tester was not able to execute the test and it needs updating.

Results of others

On the summary line of a test, an icon will appear if another tester has already executed that test in the same environment. If this icon shows, it will display the status the other user gave it (passed, failed or invalid). Hovering your mouse over the icon will display any comments the user made on invalid or failed tests. If you click this button, a new tab will open to show you the specifics of all results given for this test.

Updating a test

Click the **Edit Case Details** in the test description to update the test case. This will take you to the edit page for the test case. When you return to the run page, you will need to refresh your page to see the updates.

2.2.16 Data Import Formats

Note: Imported data should always be UTF-8 encoded.

JSON

JSON is a great way to import more complex sets of *cases* and *suites* for your product. One JSON file will be used per *product version*. Simply use the user interface to create the *Product* and *Version* that applies to the *cases* and *suites* to be imported. Then just import your JSON file to that *product version*.

Simple Example:

```
{
    "suites": [
        {
            "name": "suite name",
            "description": "suite description"
        }
    ],
    "cases": [
        {
            "name": "case title",
            "description": "case description",
            "tags": ["tag1", "tag2", "tag3"],
            "suites": ["suite1 name", "suite2 name", "suite3 name"],
            "created_by": "cdawson@mozilla.com",
            "steps": [
                {
                    "instruction": "instruction text",
                    "expected": "expected text"
                },
                {
                    "instruction": "instruction text",
                    "expected": "expected text"
                }
            ]
        }
    ]
}
```

Both top-level sections ("suites" and "cases") are optional. However, if either section is included, each item requires a "name" field value. Other than that, all fields are optional.

Importing

Importing test cases with this method involves the use of a `management command`. Before you import the cases, you must create your Product and *product version* in the user interface as mentioned above. If the *suites* in your JSON file do not already exist, they will be created for you.

Import command

Importing involves a `management command` on the command line. For this example, we are importing test cases from a file called `MyCases.json` to version `1.0` of product `Foo`. .

1. cd into your MozTrap directory
2. `./manage.py import Foo 1.0 MyCases.json`

That should be it. Now go back to the web interface and your cases will be imported.

CSV (future)

When importing from a spreadsheet or wiki set of test cases, this may prove a very useful format. This doesn't handle multiple separate steps in test cases. Rather, it presumes all steps are in a single step when imported to MozTrap.

2.2.17 Bulk Test Case Entry Formats

Gherkin-esque

This is one of the test case formats supported in the bulk test case creator.

Format:

```
Test that <test title>
<description text>
When <instruction>
Then <expected result>
```

Example:

```
Test that I can write a test
This test tests that a user can write a test
When I execute my first step instruction
then the expected result is observed
And when I execute mysecond step instruction
Then the second step expected result is observed
```

Markdown (future)

This will be another format for the bulk test case creator.

Example:

```
Test case 1 title here
======================
Description text here

* which can contain bullets
* **with formatting**
   * indentation
   * [and links](www.example.com)

Steps
-----
1. Step 1 action
    * Step 1 Expected Result
```

```
2. Step 2 action
    * Step 2 Expected Result

Test case 2 title here
======================
...
```

2.3 Installation

2.3.1 Quickstart

MozTrap requires Python 2.6 or 2.7 and MySQL 5.1+ with the InnoDB backend.

These steps assume that you have git, virtualenv, virtualenvwrapper, and a compilation toolchain available (with the Python and MySQL client development header files), and that you have a local MySQL server running which your shell user has permission to create databases in. See the full *Installation* documentation for details and troubleshooting.

1. `git clone --recursive git://github.com/mozilla/moztrap`
2. `cd moztrap`
3. `mkvirtualenv moztrap`
4. `bin/install-reqs`
5. `echo "CREATE DATABASE moztrap CHARACTER SET utf8" | mysql`
6. *create a local.py*
7. `./manage.py syncdb --migrate`
8. `./manage.py create_default_roles`
9. `./manage.py runserver`
10. Visit http://localhost:8000 in your browser.

Congratulations! If that all worked, you have a functioning instance of MozTrap for local testing, experimentation, and *development*.

Please read the *Deployment* documentation for important security and other considerations before deploying a public instance of MozTrap.

2.3.2 Detailed Install

First, clone the MozTrap repository.

Dependency source distribution tarballs are stored in a git submodule, so you either need to clone with the `--recursive` option, or after cloning, from the root of the clone, run:

```
git submodule init; git submodule update
```

If you want to run the latest and greatest code, the default `master` branch is what you want. If you want to run a stable release branch, switch to it now:

```
git checkout 1.4.5.5
```

2.3.3 Install the Python dependencies

If you want to run this project in a virtualenv to isolate it from other Python projects on your system, create the virtualenv and activate it. Then run `bin/install-reqs` to install the dependencies for this project into your Python environment.

Note: On some linux flavors, you may need to run `sudo apt-get install libmysqlclient-dev` prior to bin/install-reqs.

Installing the dependencies requires pip 1.0 or higher. pip is automatically available in a virtualenv; if not using virtualenv you may need to install it yourself.

A few of MozTrap's dependencies include C code and must be compiled. These requirements are listed in `requirements/compiled.txt`. You can either compile them yourself (the default option) or use pre-compiled packages provided by your operating system vendor.

Compiling

By default, `bin/install-reqs` installs all dependencies, including several that require compilation. This requires that you have a working compilation toolchain (`apt-get install build-essential` on Ubuntu, Xcode on OS X). It also requires the Python development headers (`apt-get install python-dev` on Ubuntu) and the MySQL client development headers (`apt-get install libmysqlclient-dev` on Ubuntu).

If you are lacking the Python development headers, you will get the error `Python.h: No such file or directory`. If you are lacking the MySQL client development files, you will get an error that `mysql_config` cannot be found.

Using operating system packages

If you prefer to use pre-compiled operating system vendor packages for the compiled dependencies, you can avoid the need for the compilation toolchain and header files. In that case, you need to install MySQLdb, py-bcrypt, and coverage (the latter only if you want test coverage data) via operating system packages (`apt-get install python-mysqldb python-bcrypt python-coverage` on Ubuntu).

If using a virtualenv, you need to ensure that it is created with access to the system packages. In virtualenv versions prior to 1.7 this was the default, in recent versions use the `--system-site-packages` flag when creating your virtualenv.

Once you have the compiled requirements installed, install the rest of the requirements using `bin/install-reqs pure`; this installs only the pure-Python requirements and doesn't attempt to compile the compiled ones. Alternatively, you can skip `bin/install-reqs` entirely and use the provided *Vendor library*.

2.3.4 Create a database

You'll need a MySQL database. If you have a local MySQL server and your user has rights to create databases on it, just run this command to create the database:

```
echo "CREATE DATABASE moztrap CHARACTER SET utf8" | mysql
```

(If you are sure that UTF-8 is the default character set for your MySQL server, you can just run `mysqladmin create moztrap` instead).

If you get an error here, your shell user may not have permissions to create a MySQL database. In that case, you'll need to append `-u someuser` to the end of that command, where `someuser` is a MySQL user who does have

permission to create databases (in many cases `-u root` will work). If you have to use `-u` to create the database, then before going on to step 5 you'll also need to create a `moztrap/settings/local.py` file (copy the sample provided at `moztrap/settings/local.sample.py`), and uncomment the `DATABASES` setting, changing the `USER` key to the same username you passed to `-u`.

2.3.5 Create the database tables

Run `./manage.py syncdb --migrate` to install the database tables.

2.3.6 Create the default user roles

This step is not necessary; you can create your own user roles with whatever sets of permissions you like. But to create a default set of user roles and permissions, run `./manage.py create_default_roles`.

2.3.7 Run the development server

Run `./manage.py runserver` to run the local development server. This server is a development convenience; it's inefficient and probably insecure and should not be used in production.

2.3.8 All done!

You can access MozTrap in your browser at http://localhost:8000.

For a production deployment of MozTrap, please read the *Deployment* documentation for important security and other considerations.

For notes on upgrading to a more recent MozTrap, see the *Upgrading* documentation.

2.4 Upgrading

To upgrade, simply use git to pull in the newer code from the GitHub repository and update the submodules:

```
git pull
git submodule update
```

If you are on a stable release branch (e.g. `0.8.X`) and you want to update to a newer release branch (e.g. `0.9.X`), make sure you've fetched the latest code on all branches, switch to the branch you want, and update to the correct version of the submodules for that branch:

```
git fetch
git checkout 0.9.X
git submodule update
```

2.4.1 Updating dependencies

Run `git submodule update` to get the latest version of the dependency submodules, and then `bin/install-reqs` to install them into your environment. Both of these commands are idempotent; there's no harm in running them every time, whether there have been any dependency changes or not.

If you are using the *Vendor library*, `bin/install-reqs` is not necessary, the submodule update will get the latest version of the vendored dependencies.

2.4.2 Database migrations

It's possible that the changes you pulled in may have included one or more new database migration scripts. To run any pending migrations:

```
python manage.py syncdb --migrate
```

This command is idempotent, so there's no harm in running it after every upgrade, whether it's necessary or not.

Warning: It is possible that a database migration will include the creation of a new database table. If you've commented out the `SET storage_engine=InnoDB init_command` in your `moztrap/settings/local.py` for performance reasons (see *Database performance tweak*), you should uncomment it before running any migrations, to ensure that all new tables are created as `InnoDB` tables.

2.5 Development

2.5.1 Coding Standards

- Python
  - Testing
  - Style
    - Line length
    - Docstrings
    - Imports
    - Whitespace
    - Line continuations
    - Comments
    - Quotes
- Javascript

Python

Testing

All tests should pass, and 100% line and branch test coverage should be maintained, at every commit (on the master branch or a release branch; temporary failing tests or lack of coverage on a feature branch is acceptable, but the branch should meet these standards before it is merged.)

To check coverage, run `bin/test` and load `htmlcov/index.html` in your browser.

Test methods should set up preconditions for a single action, take that action, and check the results of that single action (generally, separate these three blocks in the test method with blank lines). Multiple asserts in a single test method are acceptable only if they are checking multiple aspects of the result of a single action (even in that case, multiple test methods may be better unless the aspects are closely related). Avoid multi-step tests; they should be broken into separate tests.

Avoid importing the code under test at module level in the test file; instead, import it in helper methods that are called by the tests that use it. This ensures that even broken imports cause only the affected tests to fail, rather than the entire test module.

Prefer helper methods to `TestCase.setUp` for anything beyond the most basic setup (e.g. creating a user for authenticated-view tests); this keeps the setup more explicit in the test, and avoids doing unnecessary setup if not all test methods require exactly the same setup.

Never use external data fixtures for test data; use the object factories in `tests.factories` (available as `self.F` on every `tests.cases.DBTestCase`.) If a large amount of interconnected data is needed, write helper methods. External data fixtures introduce unnecessary dependencies between tests and are difficult to maintain.

Style

A consistent coding style helps make code easier to read and maintain. Many of these rules are a matter of preference and an alternate choice would serve equally well, but follow them anyway for the sake of consistency within this codebase.

If in doubt, follow **PEP 8**, Python's own style guide.

Line length Limit all lines to a maximum of 79 characters.

Docstrings Follow **PEP 257**. Every module, class, and method should have a docstring. Every docstring should begin with a single concise summary line (that fits within the 79-character limit). If the summary line is the entire docstring, format it like this:

```
def get_lib_dir():
    """Return the lib directory path."""
```

If there are additional explanatory paragraphs, place both the opening and closing triple-quotes on their own lines. Separate paragraphs with blank lines, and add an additional blank line before the closing triple quote:

```
def get_lib_dir():
    """
    Return the lib directory path.

    Checks the ``LIB_DIR`` environment variable and the ``lib-dir`` config
    file option before falling back to the default.

    """
```

Docstrings should be formatted using reStructuredText. This means that literals should be enclosed in double backticks, and literal blocks indented and opened with a double colon.

Always use triple double-quotes for enclosing docstrings.

Imports Outside of test code, prefer module-level imports to imports within a function or method. If the latter are necessary to avoid circular imports, consider reorganizing the dependency hierarchy of the modules involved to avoid the circular dependency.

Module-level imports should all occur at the top of the module, prior to any other code in the module. The following types of imports should appear in the following order (omitted if not present), each group of imports separated from the next by a single blank line:

1. Python standard library imports.
2. Django core imports.
3. Django contrib imports.
4. Other third-party module imports.

5. Imports from other modules in MozTrap.

Within each group, order imports alphabetically.

For imports from within MozTrap, use explicit relative imports for imports from the same package or the parent package (i.e. where the explicit relative import path begins with one or two dots). For more distant imports, it's usually more readable to give the full absolute path. Thus, for code in `moztrap.view.manage.runs.views`, you could do `from .forms import AddRunForm` and `from ..cases.forms import AddCaseForm`, but it's probably better to do `from moztrap.view.lists import decorators` rather than `from ....lists import decorators`; more than two dots become difficult to distinguish visually.

Never use implicit relative imports; if an import does not begin with a dot, it should be a top-level module. In other words, if `models.py` is a sibling module, always `from . import models`, never just `import models`.

Whitespace Use four-space indents. No tabs.

Strip all trailing whitespace. Configure your editor to show trailing whitespace, or automatically strip it on save. `git diff --check` will also warn about trailing whitespace.

Empty lines consisting of only whitespace are also considered "trailing whitespace". Empty lines should *not* be "indented" with trailing whitespace to match surrounding code indentation.

Separate classes and module-level functions with three blank lines. Separate class methods with two blank lines. Single blank lines may be used within functions and methods to logically group lines of code.

Line continuations Never use backslash line continuations, use Python's implicit line continuations within brackets/braces/parentheses. If necessary, prefer extraneous grouping parentheses to a backslash continuation.

All indents should be exactly four spaces.

The first place to wrap a long line is immediately after the first opening parenthesis, brace or bracket:

```
foo.some_long_method_name(
    arg_one, arg_two, arg_three, keyword="arg")

my_dict = {
    "foo": "bar", "boo": "baz"}

my_list_comprehension = [
    x[0] for x in my_list_of_tuples]
```

If the second line is still too long, each element/argument should be placed on its own line. All lines should include a trailing comma, and the closing brace/paren should go on its own line. (This allows easy rearrangement or addition/removal of items with full-line cut/paste). For example:

```
foo.some_long_method_name(
    foo=foo_arg,
    bar=bar_arg,
    baz=baz_arg,
    something_else="foo",
    )

my_dict = {
    "foo": "bar",
    "boo": "baz",
    "something else": "foo",
    }

my_list_comprehension = [
```

```
    x[0] for x in my_list_of_tuples
    if x[1] is not None
    ]
```

One exception to the four-space indents rule is when a line continuation occurs in an `if` test or another block-opening clause. In this case, indent the hanging lines eight spaces to avoid visual confusion between the line continuations and the start of the code block:

```
if (something and
        something_else and
        something_else_again):
    do_something()
```

Comments Code comments should not be used excessively; they require maintenance just as code (an out-of-date comment is often far worse than no comment at all). Comments should add information or context or rationale to the code, not simply restate what the code is doing.

The need for a comment sometimes indicates code that is overly clever or doing something unexpected. Consider whether the code should be expanded for clarity, or the API improved so the behavior is less surprising, before adding a comment.

Use `@@@` in a comment to mark code that requires future attention. This marker should always appear with explanation of why more attention is needed, or what is missing from the current code.

Quotes Always use double-quotes for quoting string literals, unless the quoted string must contain a double-quote character. Quoting such a string with single quotes is preferable to using backslash escapes in the string.

Javascript

Javascript code should pass JSLint.

The *Upgrading* documentation is also applicable to updating your development checkout of MozTrap.

2.5.2 Community

To connect with MozTrap development, visit the `#moztrap` IRC channel at `irc.mozilla.org`, or see the Pivotal Tracker backlog.

2.5.3 Updating this documentation

MozTrap documentation is hosted on ReadTheDocs.org and is maintained in the MozTrap repo. So updating the docs involves forking the repo, changing the appropriate reStructuredText documents and submitting a pull request. Then the team will review them and merge them after any needed adjustments are made.

So here are your steps:

1. fork the MozTrap repo
2. make any changes in the `/docs` folder using Sphinx and reStructuredText formatting
3. test that your changes are correctly formatted by installing the python Sphinx package (in the repo's `requirements.txt` document) by typing `make html` in that same `/docs` folder
4. load the file: `/docs/_build/html/index.html` into your browser (it's Firefox, right?) to test your changes

5. submit your pull request and it will be reviewed shortly
6. receive a big thanks for helping!!

2.5.4 Coding standards

See the *Coding Standards* for help writing code that will maintain a consistent style and quality with the rest of the codebase.

2.5.5 User registration

MozTrap's default settings use Django's "console" email backend to avoid requiring an SMTP server or sending real emails in development/testing mode. So when registering a new user, pay attention to your runserver console; this is where the confirmation email text will appear with the link you need to visit to activate the new account.

2.5.6 Running the tests

To run the tests, after installing all Python requirements into your environment:

```
bin/test
```

To view test coverage data, load `htmlcov/index.html` in your browser after running the tests.

To run just a particular test module, give the dotted path to the module:

```
bin/test tests.model.core.models.test_product
```

Give a dotted path to a package to run all tests within that package, including in submodules:

```
bin/test tests.model.core
```

2.5.7 Compass/Sass

MozTrap's CSS (located in *static/css*) is generated using Sass and the Compass framework, with the Susy grid plugin. Sass source files are located in *sass/*.

The generated CSS is included with MozTrap, so Sass and Compass are not needed to run MozTrap. You only need them if you plan to modify the Sass sources and re-generate the CSS.

To install the necessary Ruby gems for Compass/Sass development, run `bin/install-gems`. Update `requirements/gems.txt` if newer gems should be used.

While tweaking the sass files, you should run the command line file to update the CSS as you go. To do this:

```
compass watch
```

or a workaround to a bug for Mac OS 10.8:

```
compass watch --poll
```

2.5.8 Loading sample data

A JSON fixture of sample data is provided in `fixtures/sample_data.json`. To load this fixture, run `bin/load-sample-data`.

> **Warning:** Loading the sample data will overwrite existing data in your database. Do not load it if you have data in your database that you care about.

The sample data already includes the *default roles*, so there is no need to run a separate command to create them.

The sample data also includes four users, one for each default role. Their usernames are *tester*, *creator*, *manager*, and *admin*. All of them have the password `testpw`.

Resetting your database

To drop your database and create a fresh one including only the sample data, run these commands:

Note: If your shell user doesn't have the MySQL permissions for the first two commands, you may need to append e.g. *-uroot* to them.

```
mysqladmin drop moztrap
echo "CREATE DATABASE moztrap CHARACTER SET utf8" | mysql
python manage.py syncdb --migrate
bin/load-sample-data
```

If you create a superuser during the course of the `syncdb` command (recommended so that you can access the Django admin), the sample data fixture will not overwrite that superuser.

Regenerating the sample data

The sample data fixture is generated using django-fixture-generator via the code in `moztrap/model/core/fixture_gen.py`, `moztrap/model/environments/fixture_gen.py`, `moztrap/model/tags/fixture_gen.py`, `moztrap/model/library/fixture_gen.py` and `moztrap/model/execution/fixture_gen.py`.

If you've modified one of the above files, you can regenerate the fixture by running `bin/regenerate-sample-data`.

2.5.9 Adding or updating a dependency

Adding a new dependency (or updating an existing one to a newer version) involves a few steps, since the requirements files and both submodules (the requirements tarballs submodule in `requirements/dist` and the *Vendor library* submodule in `requirements/vendor`) must be updated.

Preparing your checkout

By default, the submodules are both checked out via a read-only anonymous URL, so that anyone can check them out. In order to push commits to the submodules, you'll need to switch the push url to use ssh. Make this change as follows:

```
cd requirements/dist
git remote set-url --push origin git@github.com:mozilla/moztrap-reqs

cd ../vendor
git remote set-url --push origin git@github.com:mozilla/moztrap-vendor-lib
```

This assumes that you have permission to push to the primary `moztrap-reqs` and `moztrap-vendor-lib` repositories. If instead you have made your own forks of one or both of these repositories, adjust the above URLs to push to your fork.

Adding the dependency tarball

Assuming the new dependency is a Python package available on PyPI (for the sake of this example we'll assume that we want the 2.1.1 version of the Markdown package), from the root of your MozTrap checkout run this command in order to download the tarball into `requirements/dist`:

```
pip install -d requirements/dist Markdown==2.1.1
```

This should add the `Markdown-2.1.1.tar.gz` file into `requirements/dist`. We want to add this file and commit the change to the submodule. First, though, we need to ensure that we are actually committing on a branch in the submodule, since by default git does not check out submodules on a branch.

In most cases, you can just check out the `master` branch of the submodule and commit there:

```
cd requirements/dist
git checkout master
git add Markdown-2.1.1.tar.gz
# "git rm" the older Markdown tarball, if you're updating
git commit -m "Add Markdown 2.1.1."
git push
```

Note: If you are working on a release branch of MozTrap rather than the master branch, you may find that updating the submodule to `master` updates the version of some dependency to a more recent version, and your branch of MozTrap is not prepared for this dependency update. In that case rather than updating to the submodule's master branch, you should create a new branch of the submodule with a name matching the branch of MozTrap you are working on; replace `git checkout master` in the above with e.g. `git branch 0.8.X`. (If you've already done the `git checkout master`, go back out to the MozTrap repo root and `git submodule update` to get back to the pinned commit of the submodule, then `cd requirements/dist` and `git branch 0.8.X`.) If you create your own branch of the submodule, you may need to also replace `git push` with e.g. `git push -u origin 0.8.X`).

Similarly, if you are working on a feature branch, and your feature branch requires a newer version of a dependency, it is preferable to make a branch of the submodule. The master branch of MozTrap is tied to a specific commit of the submodule, so it won't create an immediate problem if you just push to the submodule's master branch; but if some other feature on the master branch must also update a dependency, there could be a problem if everyone is just pushing to the submodule's master branch. (If you are just adding a dependency, not changing the version of an existing one, this really isn't an issue, as having the extra tarball around won't hurt anything for another branch).

Updating the requirements file

If your added dependency is a pure-Python dependency (no compiled C extensions), add an entry to `requirements/pure.txt` like `Markdown==2.1.1`.

If your added dependency does require compilation, add it to `requirements/compiled.txt` instead.

If you are just updating the version of an existing dependency, find the existing requirement line and change the version.

Updating the vendor library

Note: This step is only necessary for pure-Python dependencies. Compiled dependencies should not be included in the vendor library.

Note: Due to a bug in pip, this step currently must be done within an empty `--no-site-packages` virtualenv. (Virtualenv 1.7+ automatically creates `--no-site-packages` envs by default; with an earlier version you must use the `--no-site-packages` flag).

If you've correctly created and activated a `-no-site-packages` virtualenv, `pip freeze` should show only the `wsgiref` package (which is part of the Python standard library).

Now, from the root of the MozTrap repo, run:

```
bin/generate-vendor-lib
cd requirements/vendor
git status
```

The only changed files shown here should be the new Python files for your added dependency (or, if upgrading a dependency, possibly some added/modified/removed files, but nothing outside the one upgraded package).

If that is the case, commit your changes to the master branch (or the branch you chose earlier) and push using the same steps as shown above for the `requirements/dist` submodule.

Pulling it all together

At this point, if you run `git status` in the root of the MozTrap repo, you should see three modifications: a modification to `requirements/pure.txt` and `(new commits)` in the `requirements/dist` and `requirements/vendor` submodules (or, if you added a compiled module, a modification to `requirements/compiled.txt` and `(new commits)` only in `requirements/dist`).

Add these changes, commit, push, and you're done!

```
git add requirements/
git ci -m "Add Markdown 2.1.1 dependency."
git push
```

2.6 Deployment

Django's built-in `runserver` is not suitable for a production deployment; use a WSGI-compatible webserver such as Apache with mod_wsgi, or gunicorn. A WSGI application callable is provided in `moztrap/deploy/wsgi.py` in the `application` object.

You'll also need to serve the static assets; Apache or nginx can do this.

You'll need a functioning SMTP server for sending user registration confirmation emails; configure the `EMAIL_*` settings and `DEFAULT_FROM_EMAIL` in your `moztrap/settings/local.py` to the appropriate values for your server.

The default local-memory cache backend is not suitable for use with a production (multi-process) webserver; you'll get CSRF errors on login because the CSRF token won't be found in the cache. You need an out-of-process cache backend: memcached or Redis is recommended for production deployment. The Django file or database cache backends may also work for a small deployment that is not performance-sensitive. Configure the `CACHE_BACKENDS` setting in `moztrap/settings/local.py` for the cache backend you want to use.

In addition to the notes here, you should read through all comments in `moztrap/settings/local.sample.py` and make appropriate adjustments to your `moztrap/settings/local.py` before deploying this app into production.

2.6.1 Logins

By default all access to the site requires authentication. If the `ALLOW_ANONYMOUS_ACCESS` setting is set to `True` in `moztrap/settings/local.py`, anonymous users will be able to read-only browse the management and test-results pages (but will not be able to submit test results or modify anything).

By default MozTrap uses BrowserID for all logins, but it also supports conventional username/password logins. To switch to username/password logins, just set `USE_BROWSERID` to `False` in `moztrap/settings/local.py`.

If using BrowserID (the default), you need to make sure that your `SITE_URL` is set correctly in `moztrap/settings/local.py`, or BrowserID logins will not work.

2.6.2 Vendor library

For deployment scenarios where pip-installing dependencies into a Python environment (as `bin/install-reqs` does) is not preferred, a pre-installed vendor library is provided in `requirements/vendor/lib/python`. This library does not include the compiled dependencies listed in `requirements/compiled.txt`; these must be installed separately via e.g. system package managers. The `site.addsitedir` function should be used to add the `requirements/vendor/lib/python` directory to sys.path, to ensure that `.pth` files are processed. A WSGI entry-point script is provided in `moztrap/deploy/vendor_wsgi.py` that makes the necessary `sys.path` adjustments, as well as a version of `manage.py` in `vendor-manage.py`.

If you are using the vendor library and you want to run the MozTrap tests, `bin/test` won't work as it uses `manage.py`. Instead run `python vendor-manage.py test`.

If you need code coverage metrics (and you have the `coverage` module installed; it isn't included in the vendor library as it has a compiled extension), use this:

```
coverage run vendor-manage.py test
coverage html
firefox htmlcov/index.html
```

2.6.3 Security

In a production deployment this app should be served exclusively over HTTPS, since almost all use of the site is authenticated, and serving authenticated pages over HTTP invites session hijacking attacks. The `SESSION_COOKIE_SECURE` setting should be set to `True` in `moztrap/settings/local.py` when the app is being served over HTTPS.

Run `python manage.py checksecure` on your production deployment to check that your security settings are correct.

2.6.4 Static assets

This app uses Django's staticfiles contrib app for collecting static assets from reusable components into a single directory for production serving, and uses django-compressor to compress and minify them. Follow these steps to deploy the static assets into production:

1. Ensure that `COMPRESS_ENABLED` and `COMPRESS_OFFLINE` are both uncommented and set to `True` in `moztrap/settings/local.py`.
2. Run `python manage.py collectstatic` to collect all static assets into the `collected-assets` directory (or whatever `STATIC_ROOT` is set to in `moztrap/settings/local.py`).
3. Run `python manage.py compress` to minify and concatenate static assets.
4. Make the entire resulting contents of `STATIC_ROOT` available over HTTP at the URL `STATIC_URL` is set to.

If deploying to multiple static assets servers, probably steps 1-3 should be run once on a deployment or build server, and then the contents of `STATIC_ROOT` copied to each web server.

2.6.5 Database performance tweak

In order to ensure that all database tables are created with the InnoDB storage engine, MozTrap's default settings file sets the database driver option "init_command" to "SET storage_engine=InnoDB". This causes the SET command to be run on each database connection, which is an unnecessary slowdown once all tables have been created. Thus, on a production server, you should comment this option from your `moztrap/settings/local.py` file's `DATABASES` setting after you've run `python manage.py syncdb --migrate` to create all tables (uncomment it before running `python manage.py syncdb` or `python manage.py migrate` after an update to the MozTrap codebase, or before trying to run the tests).

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

HTTP Routing Table

/api

GET /api/v1/<object_type>/, 29
POST /api/v1/<object_type>/, 29
PUT /api/v1/<object_type>/<id>, 29
DELETE /api/v1/<object_type>/<id>, 29
GET /api/v1/<object_type>/<id>/, 29
GET /api/v1/case, 31
POST /api/v1/case, 32
GET /api/v1/case/<id>, 31
PUT /api/v1/case/<id>, 32
DELETE /api/v1/case/<id>, 32
GET /api/v1/casestep, 33
POST /api/v1/casestep, 33
PUT /api/v1/casestep/<id>, 33
DELETE /api/v1/casestep/<id>, 33
GET /api/v1/caseversion, 32
POST /api/v1/caseversion, 32
GET /api/v1/caseversion/<id>, 32
PUT /api/v1/caseversion/<id>, 33
DELETE /api/v1/caseversion/<id>, 33
GET /api/v1/category, 36
POST /api/v1/category, 36
GET /api/v1/category/<id>, 36
PUT /api/v1/category/<id>, 36
DELETE /api/v1/category/<id>, 36
POST /api/v1/element, 37
GET /api/v1/element/, 36
GET /api/v1/element/<id>, 37
PUT /api/v1/element/<id>, 37
DELETE /api/v1/element/<id>, 37
GET /api/v1/environment, 37
PATCH /api/v1/environment, 37
POST /api/v1/environment, 37
GET /api/v1/environment/<id>, 37
PUT /api/v1/environment/<id>, 37
DELETE /api/v1/environment/<id>, 37
GET /api/v1/product, 30
POST /api/v1/product, 30
GET /api/v1/product/<id>, 30
PUT /api/v1/product/<id>, 30
DELETE /api/v1/product/<id>, 30
GET /api/v1/productversion, 31
POST /api/v1/productversion, 31
GET /api/v1/productversion/<id>, 31
PUT /api/v1/productversion/<id>, 31
DELETE /api/v1/productversion/<id>, 31
GET /api/v1/profile, 36
POST /api/v1/profile, 36
GET /api/v1/profile/<id>, 36
PUT /api/v1/profile/<id>, 36
DELETE /api/v1/profile/<id>, 36
PATCH /api/v1/result, 35
GET /api/v1/run, 34
POST /api/v1/run, 35
GET /api/v1/runcaseversion, 35
GET /api/v1/suite, 33
POST /api/v1/suite, 34
PUT /api/v1/suite/<id>, 34
DELETE /api/v1/suite/<id>, 34
GET /api/v1/suitecase, 34
POST /api/v1/suitecase, 34
GET /api/v1/suitecase/<id>, 34
PUT /api/v1/suitecase/<id>, 34
DELETE /api/v1/suitecase/<id>, 34
GET /api/v1/tag, 38
POST /api/v1/tag, 38
GET /api/v1/tag/<id>, 38
PUT /api/v1/tag/<id>, 39
DELETE /api/v1/tag/<id>, 39

Index

P

Python Enhancement Proposals
    PEP 257, 50
    PEP 8, 50