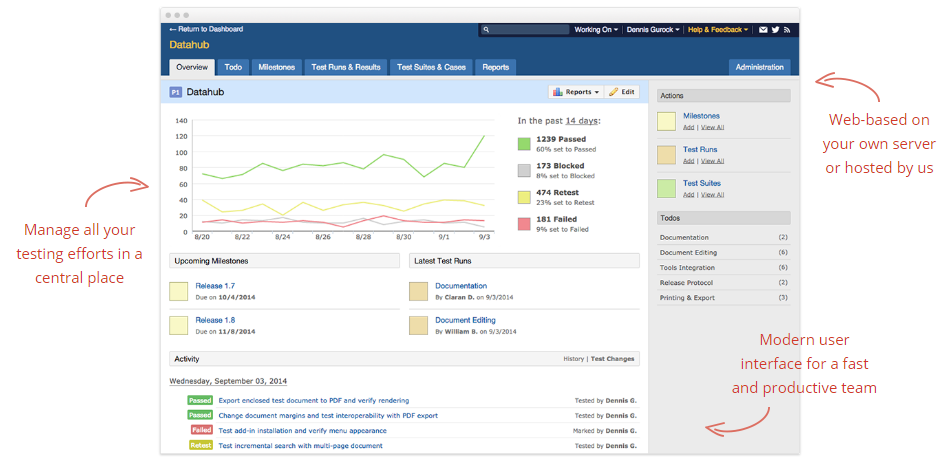
为什么要使用测试工具？

程序的测试是一项枯燥、重复的劳动，每当程序发布一个新的版本后，首先要跟踪缺陷日志，对上一个版本中的BUG进行测试、参看是否已经修正。其次还要对没有问题的功能重新测试一遍，检查是否由于程序的改动而引发了新的BUG。对于大型项目来讲，版本更新相当频繁，每当更新一个版本后就重新进行测试是一项相当繁重的任务，有时甚至当一个版本的测试用例还没有遍历完，新的一个版本就发布了，为了避免这种情况的发生，我们可以借助测试工具来帮助我们消除这些重复性的劳动。除此之外，测试工具具有更高的稳定性、精确性，不会遗漏掉任何一个测试用例，而且可以大量的节省测试的时间。

为什么要选择TestRail

1、TestRail可以为团队带来先进的测试管理：



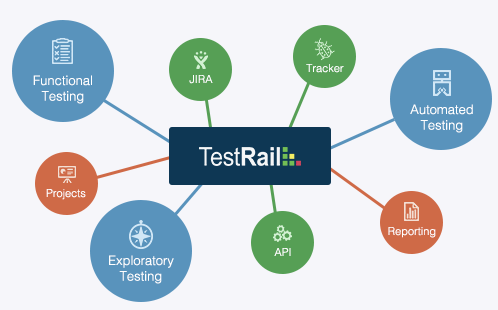
包括：

（1）一个软件就可以管理、组织和跟踪所有的测试工作

（2）软件基于web，一旦部署服务器整个团队就可以轻松访问TestRail

（3）软件拥有漂亮而高效的界面

2、TestRail非常适合敏捷和传统的测试方法



（1）适用于任何工作流：无论你采用哪种敏捷和传统的测试方法，TestRail都能胜任

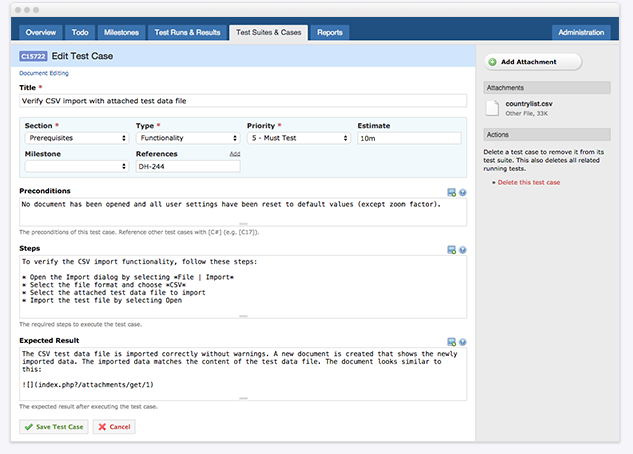
（2）适用于任何团队：1名测试人员与1000名测试人员组成的团队都可以使用TestRail

（3）适用于任何方法：TestRail可以让你轻松管理功能性的、探索性的和自动化测试

3、灵活的项目组织

TestRail可以管理所有正在进行的测试项目,可以轻松访问所有相关项目的细节，包括项目里程碑、活动测试和以往的测试结果。还可以简单地归档已完成项目，支持多个团队并行工作且对项目的进展进行跟踪。

4、有丰富且功能强大的测试用例管理



包括：

（1）丰富的测试用例可以很好地帮助团队验证应用程序的功能和需求

（2）结构化的方式可以轻松地管理项目的所有细节，包括前提条件、步骤和预期结果

（3）采用多种文字形式，例如添加附件、截图以及定制的功能等

（4）能够非常方便地管理测试套装(suites)、功能分支(sections)、测试用例(cases)

（5）有完整的历史记录可以追踪变化，保证透明性

5.TestRail还有很多特性，例如：

（1）简单执行测试样例

（2）可以轻松管理软件的里程碑、发行与迭代

（3）内置的可操作的实时度量和报告

（4）提供时间与进程预测

（5）提供自动化测试与测试API

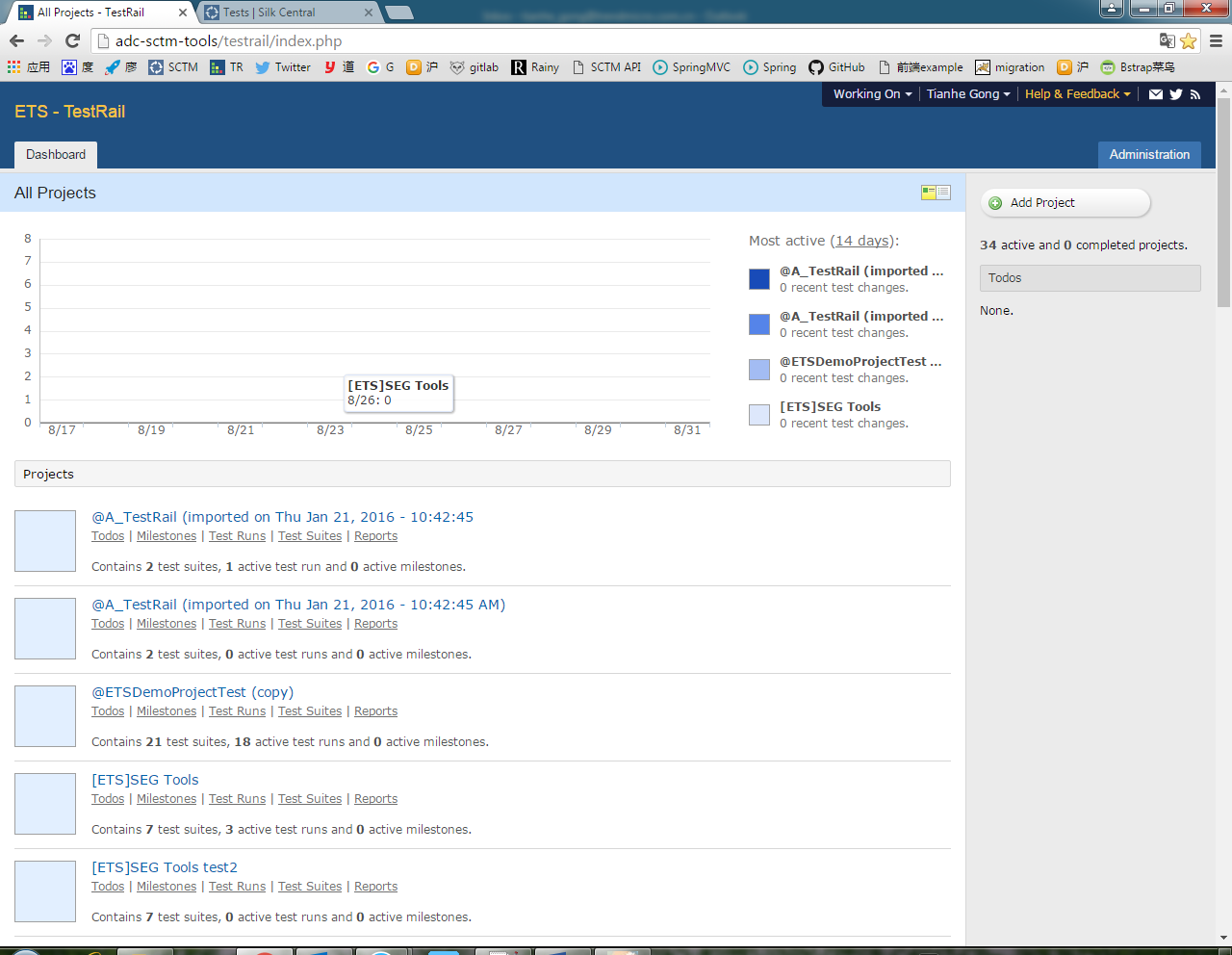
（6）涉及覆盖与需求的完整性分析

（7）非常简单地提供灵活的且定制化的需求

……

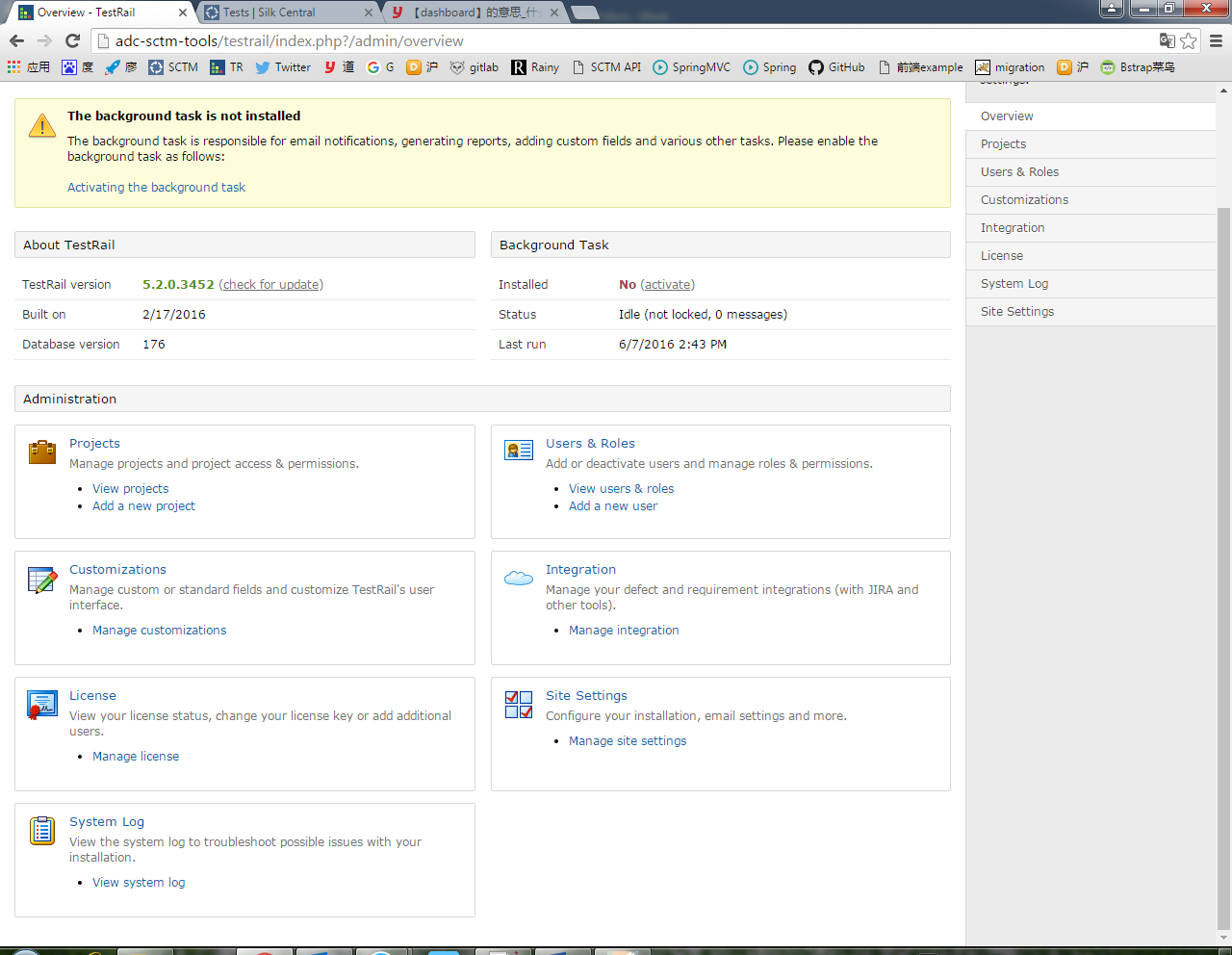
TestRail的使用

1、使用TestRail的Administrator账户创建用户

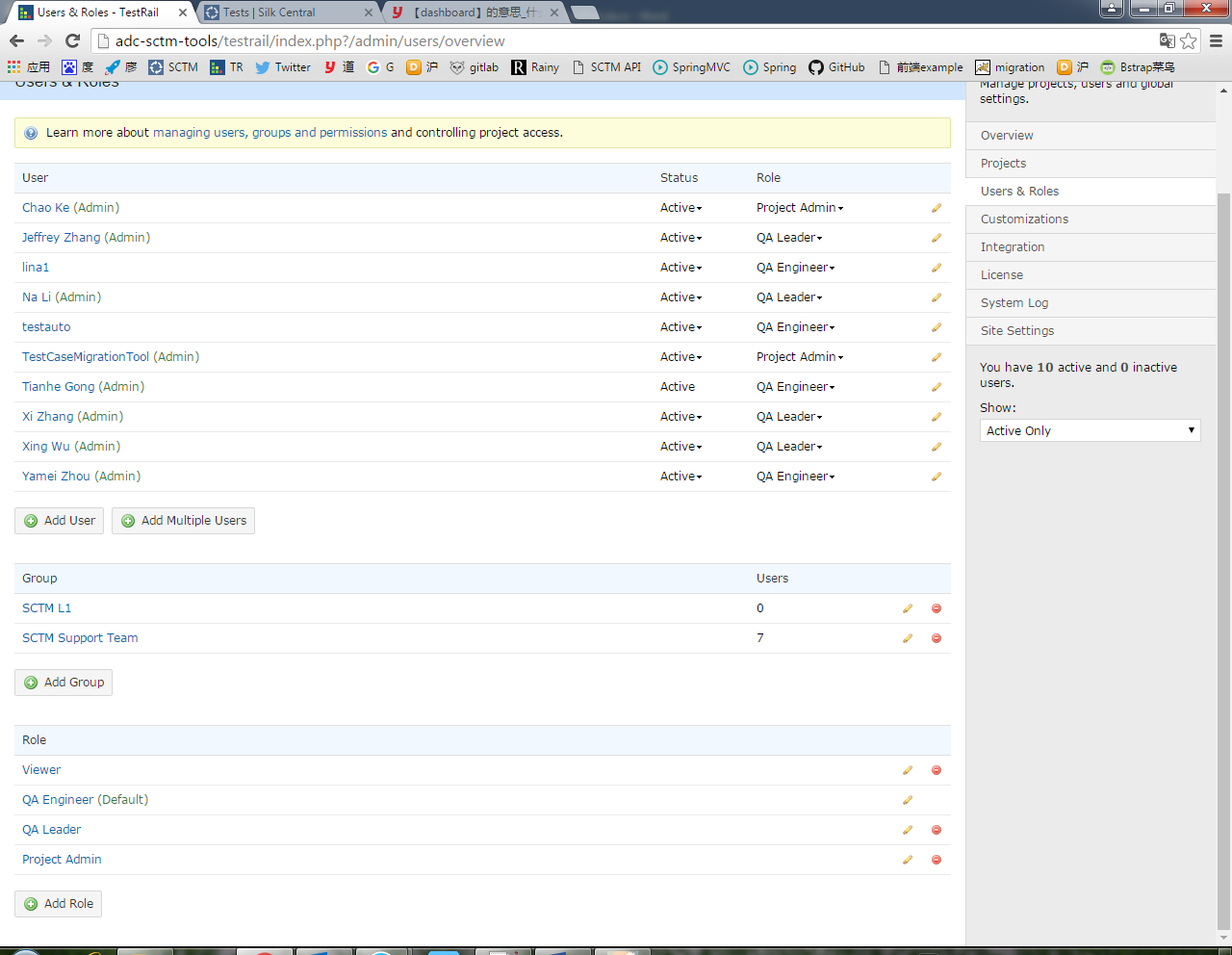


输入TestRail的URL，我们就进入了整个系统。首先出现的是TestRail的dashboard，dashboard展示了所有的项目以及近期的活动测试。

点击右上角的Administrator按钮，进入管理员页面。

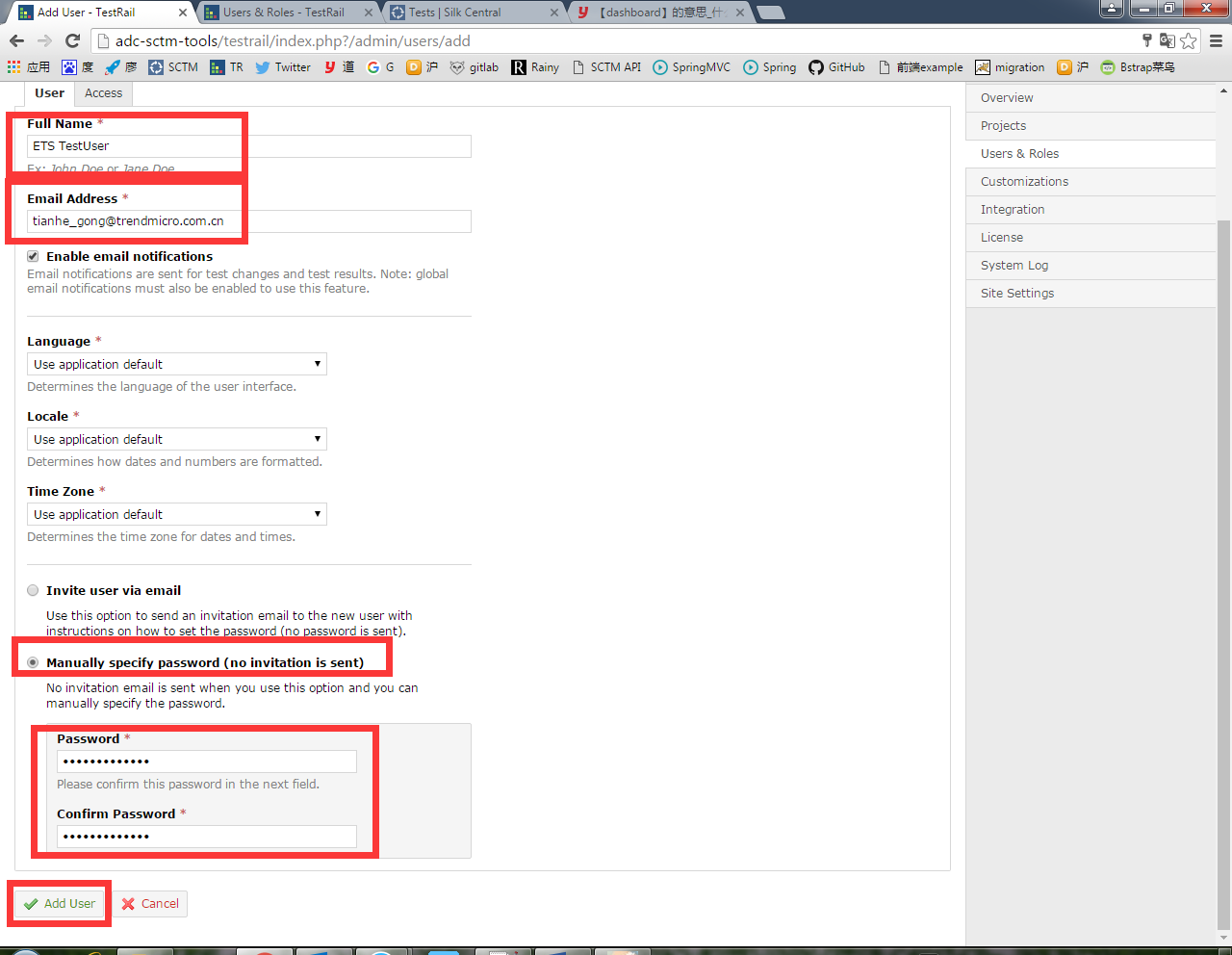


管理员页面中，我们可以对项目、用户、定制、完整性等进行配置。首先我们点击“Users & Roles”。

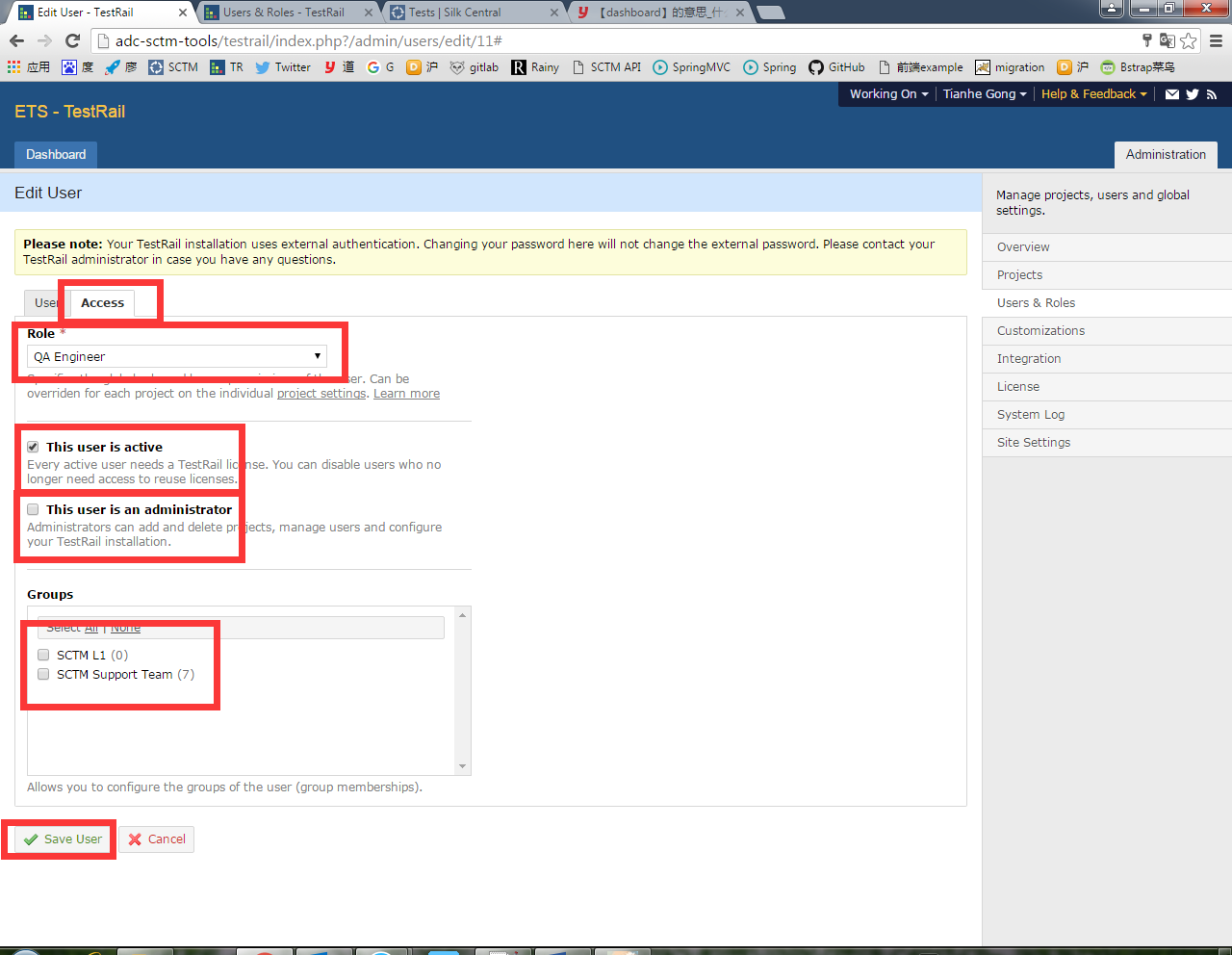


这里我们可以管理每一个用户。指定用户是否是激活的、为用户分配角色、分配分组以及设置每个角色的权限。

点击Add User按钮，进入用户添加界面。



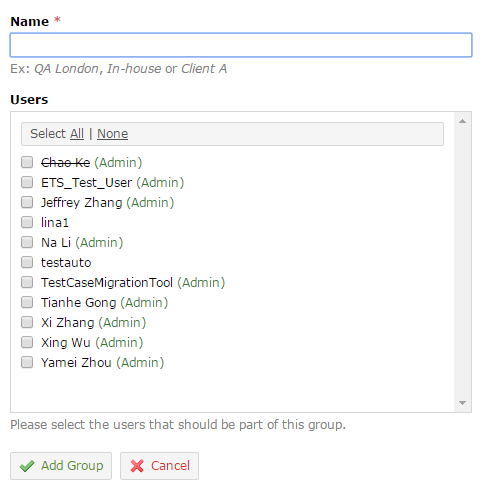
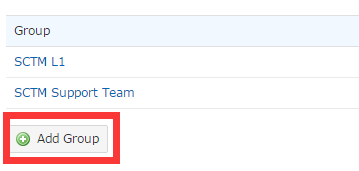
我们先后为用户指定用户名、邮箱，并手动为用户指定密码。



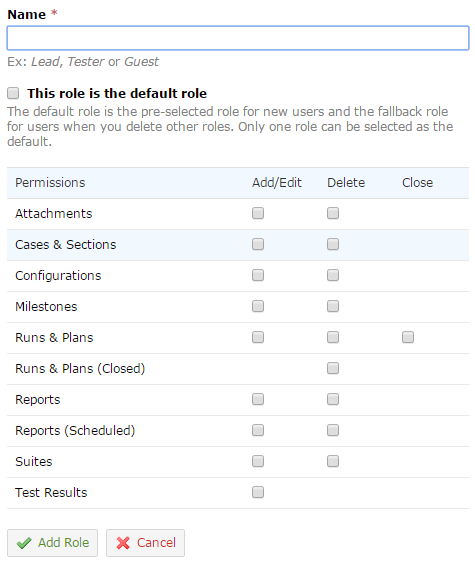
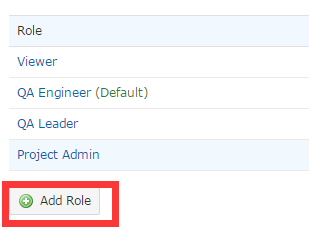
点击Access标签页，为用户分配角色、指定为激活状态并指定为管理员、分配组。最后点击 Add User按钮。



用户已经被成功地添加进了用户列表。如果以后需要修改用户信息，点击用户最右边的“铅笔”图标，即可回到之前的页面，修改完毕后点击“Save User”即可。

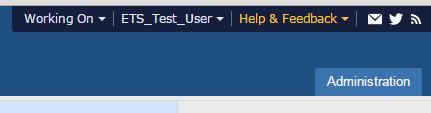
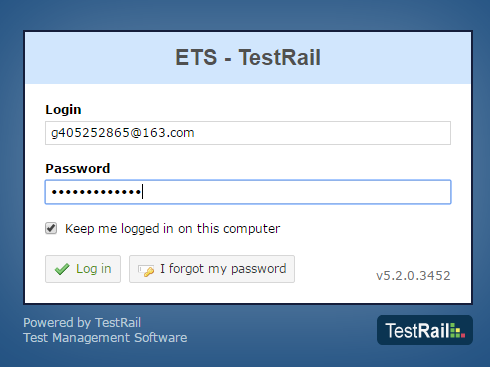


点击Add Group可以添加一个组，我们只需要为新的分组指定组名、组内成员即可。直接点击已有的组可以修改组信息。



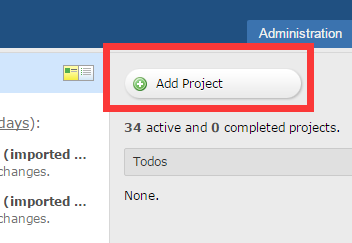
点击Add Role按钮可以添加一个角色。我们需要为该角色选择权限。点击角色名可以修改角色名称或权限的配置。

我们使用刚刚创建的ETS\_Test\_User登录系统。

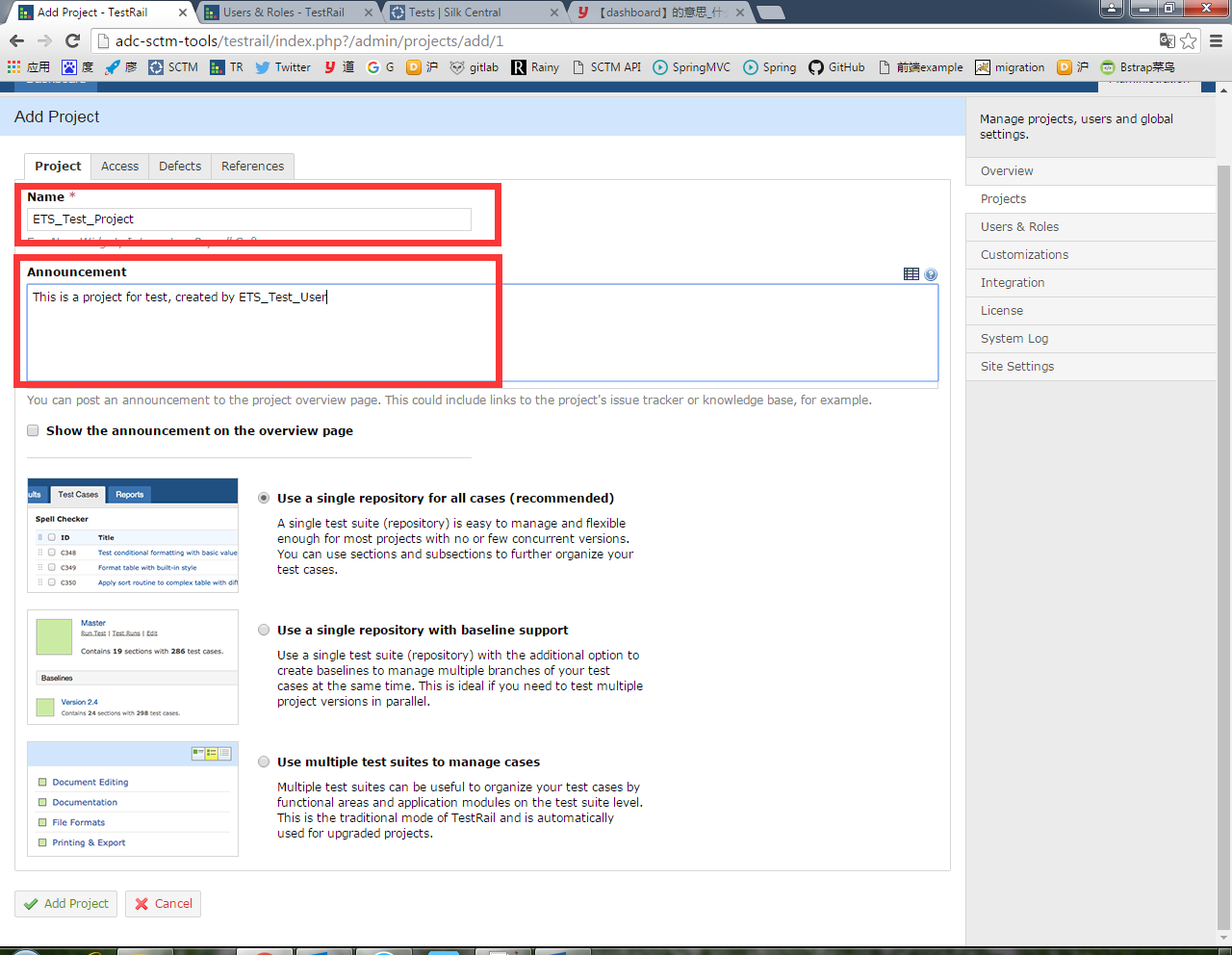


点击Log in按钮，即可登录成功。（注意LogIn内需要填写邮箱地址，而不能是Full Name）

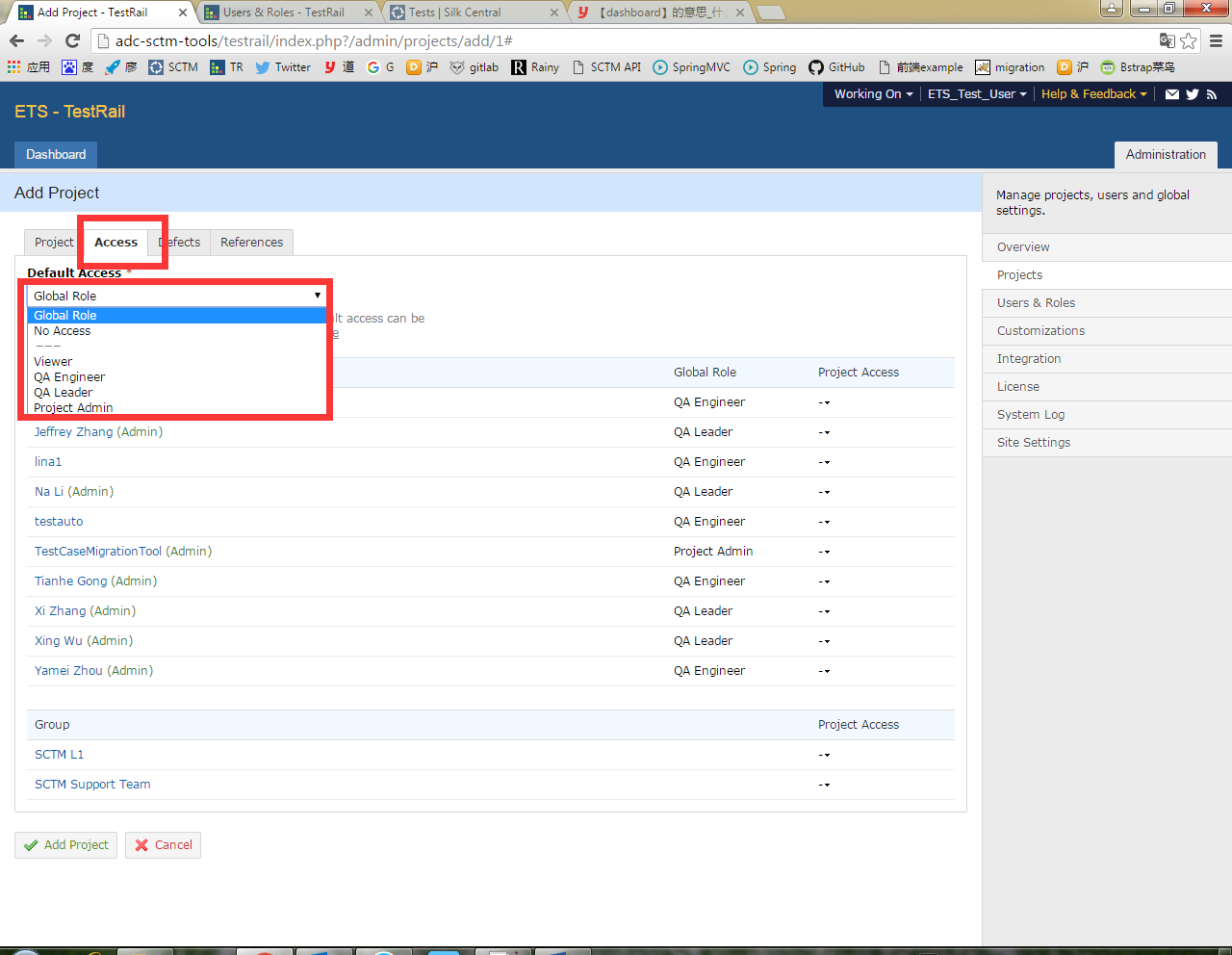
2、使用TestRail创建一个项目



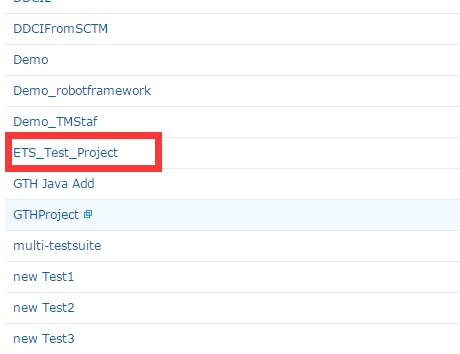
点击右上角的Add Project，进入创建项目页面。



点击Access标签页我们可以设置哪些角色可以访问到这个project



点击Add Project按钮，则项目添加成功。我们可以在项目列表中看到这个项目。



项目根据Suite的种类可以分为3种：

（1）使用一个单个的仓库（repository），里面存放所有的测试用例

（2）使用一个单个的仓库，并提供基线（baseline）的支持

（3）使用多个测试套装

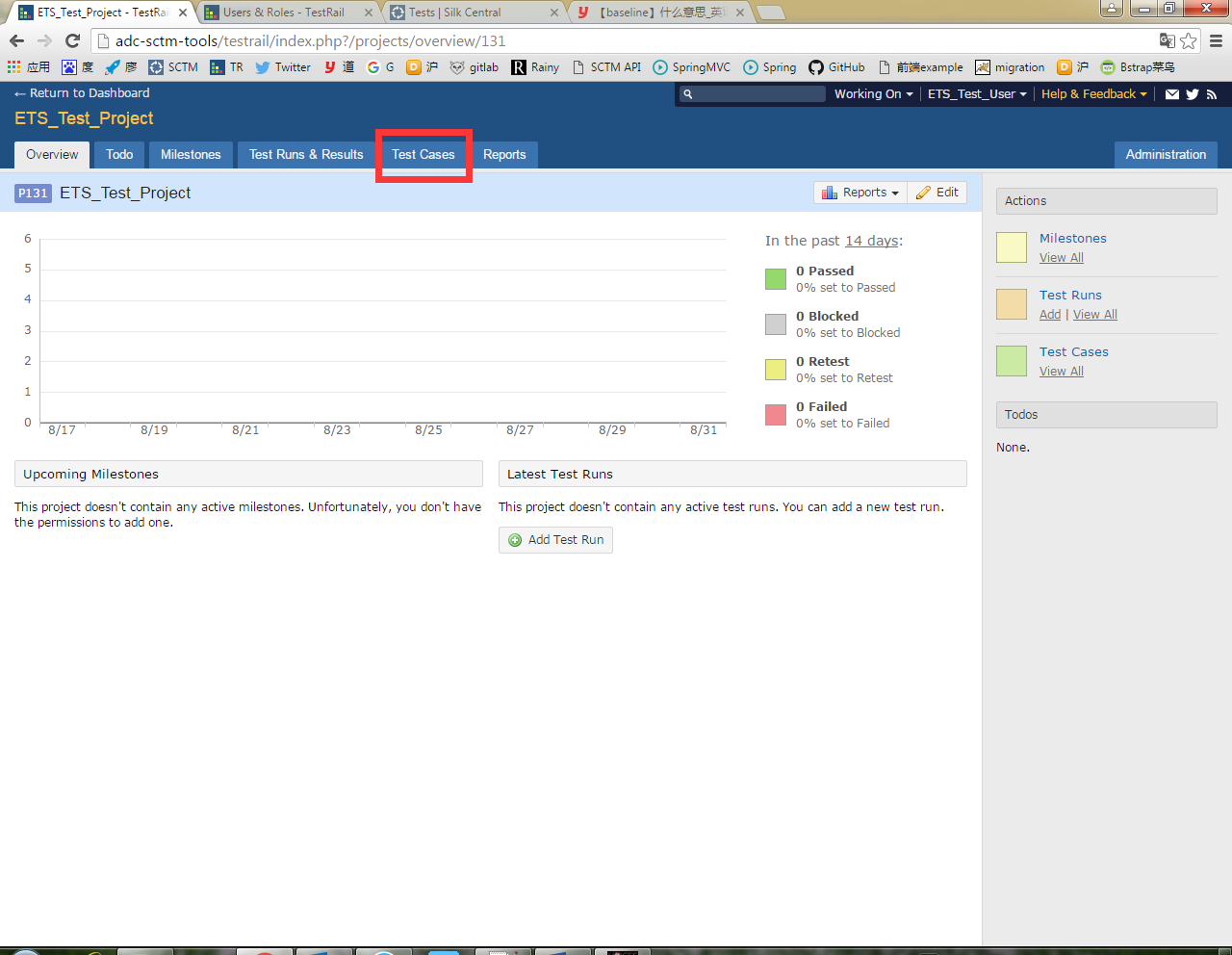
测试用例和功能分支组成了一颗树形结构。其中suite是树的根节点。在第一种情况下，只有一颗测试树，因为仅使用了一个suite节点。我们可以在suite里面建立section，或者在section的下面建立section。最后cases建立在section之上，是树的叶子节点。值得注意的是case不可以与suite直接相连。

第二种情况下，用户拥有一个主suite作为根节点（自动命名为Master）。此外，用户可以为项目添加baseline，添加后的baseline类似于suite，一般来讲baseline视为项目的分支。如果项目需要多个版本并行地去测试，则推荐使用这种选项。

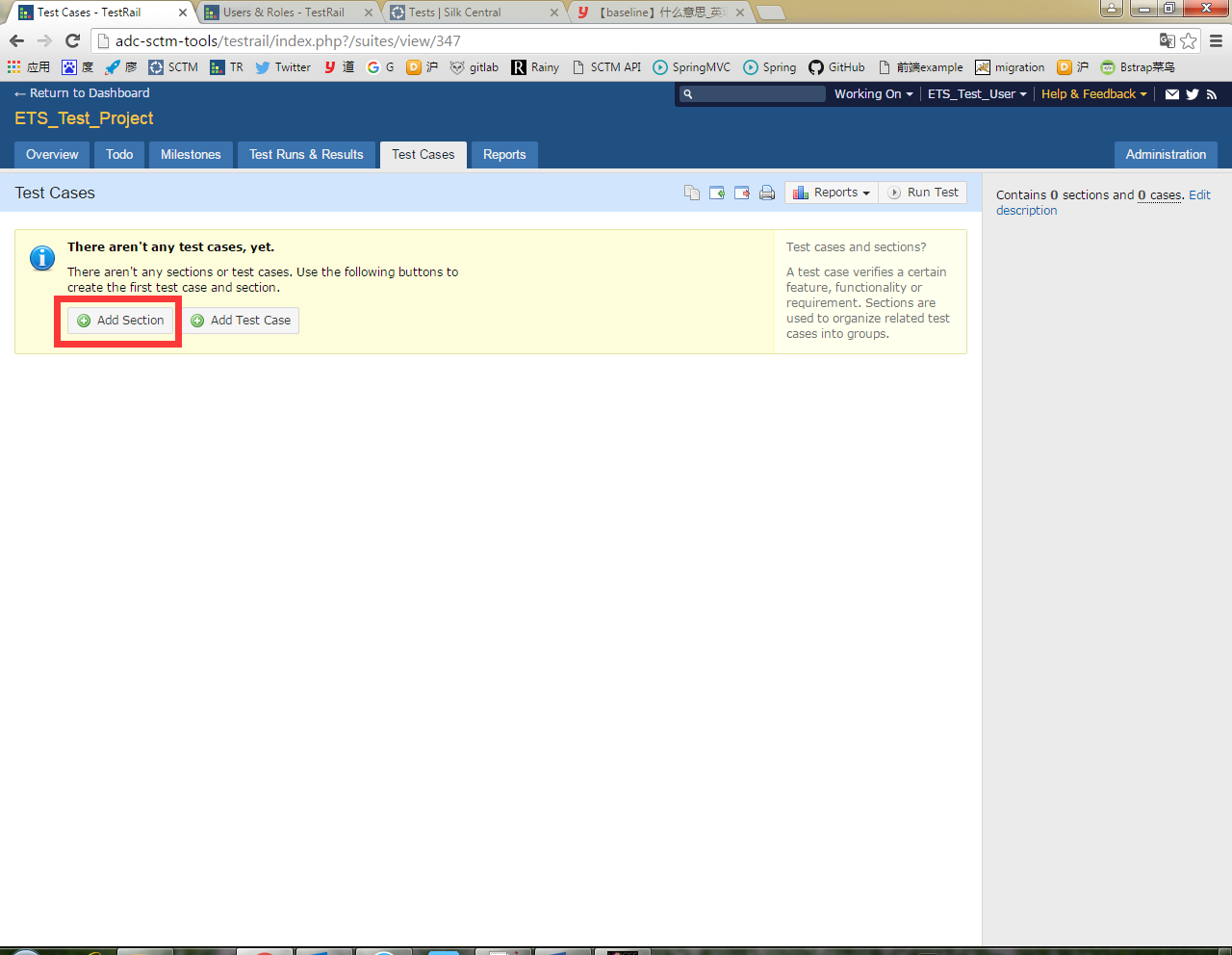
第三种情况下，用户拥有多个suite。用户可以根据功能、模块或者版本等不同将用例放在不同的suite中。这也是最传统的测试结构。

在事例中我们采用的第一种结构。下面我们建立section和case：

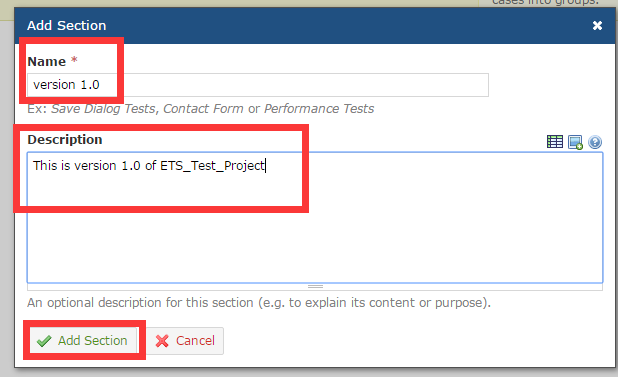
在dashboard里面点击项目名即可进入项目管理界面。



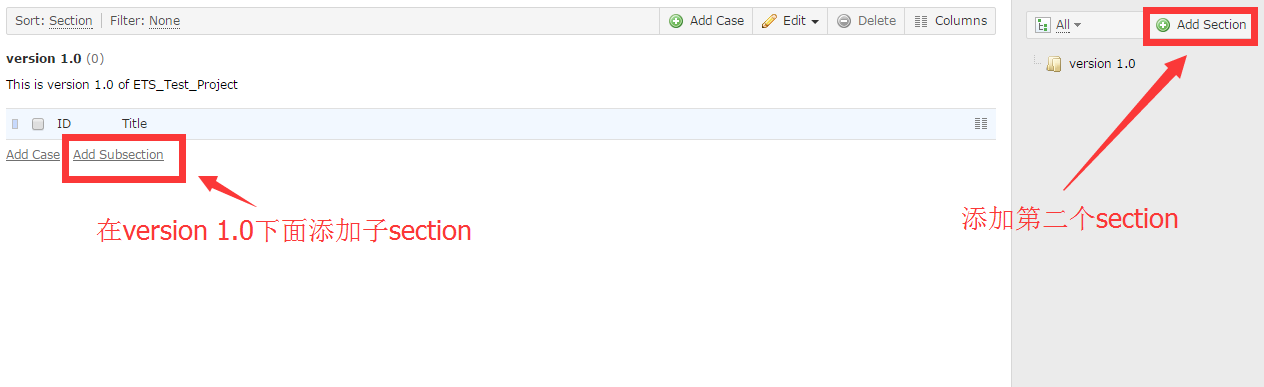
点击Test Cases标签



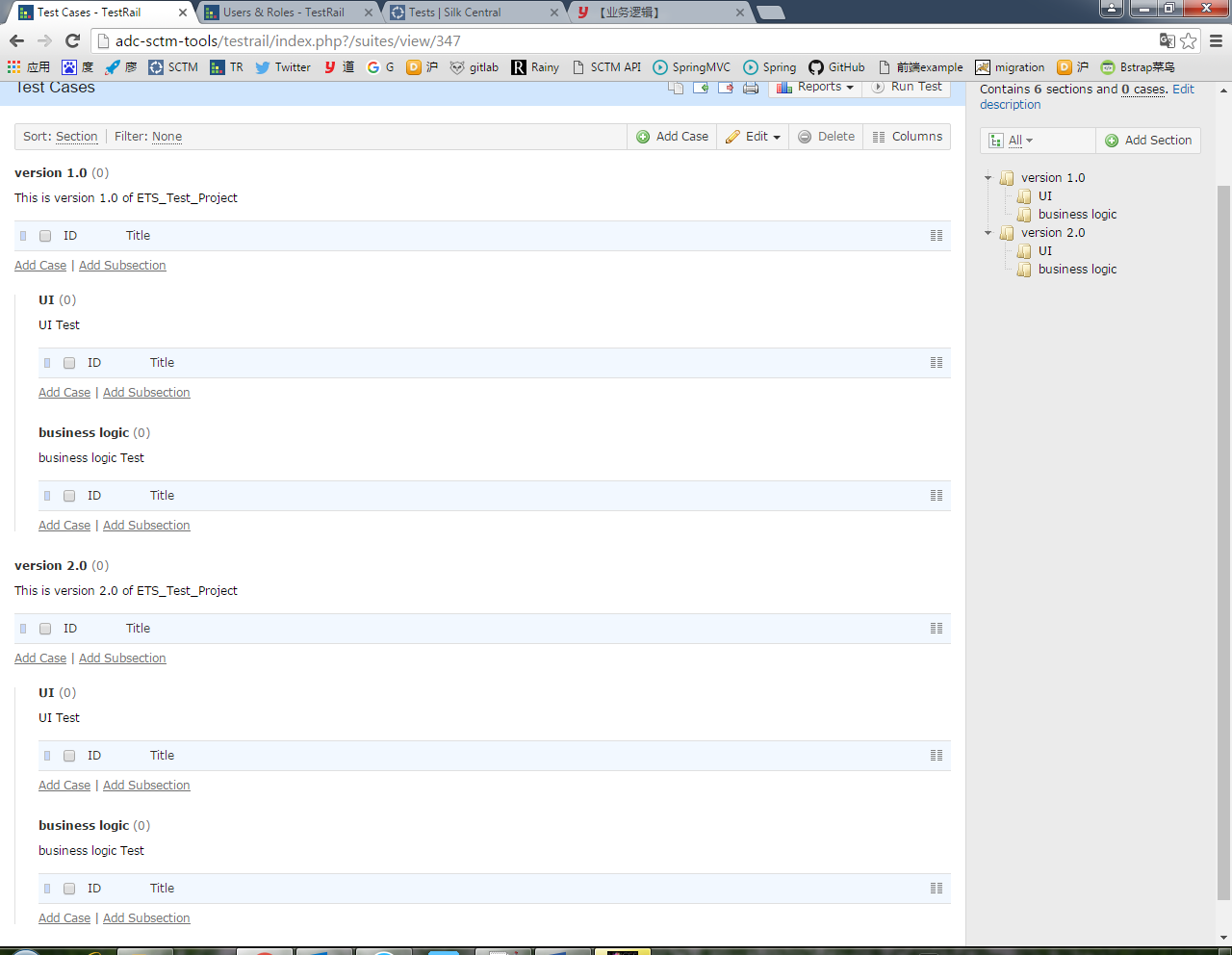
我们假设项目有2个版本：version 1.0和version 2.0。我们先分别建立两个版本的section。点击Add Section。



输入section的名字和描述后，点击Add Section即可建立section。



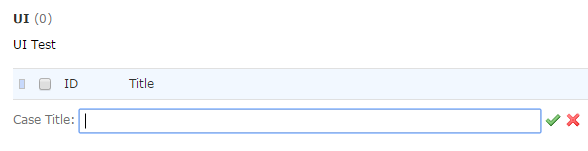
看到上图说明添加成功。我们假设section version 1.0下面我们需要对两项功能进行测试，一个是UI，一个是业务逻辑。点击Add Subsection添加子section。点击最后边的Add Section添加另一个section version 2.0。用同样的方法添加UI和业务逻辑的子section。



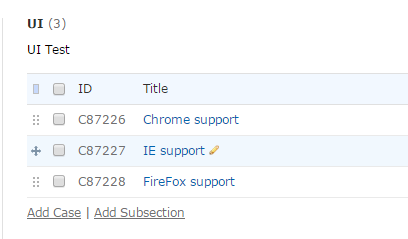
添加完之后如图所示。

下面我们为section添加case。假设我们UI测试需要测试Chrome、IE和FireFox的支持，则添加3个测试用例。

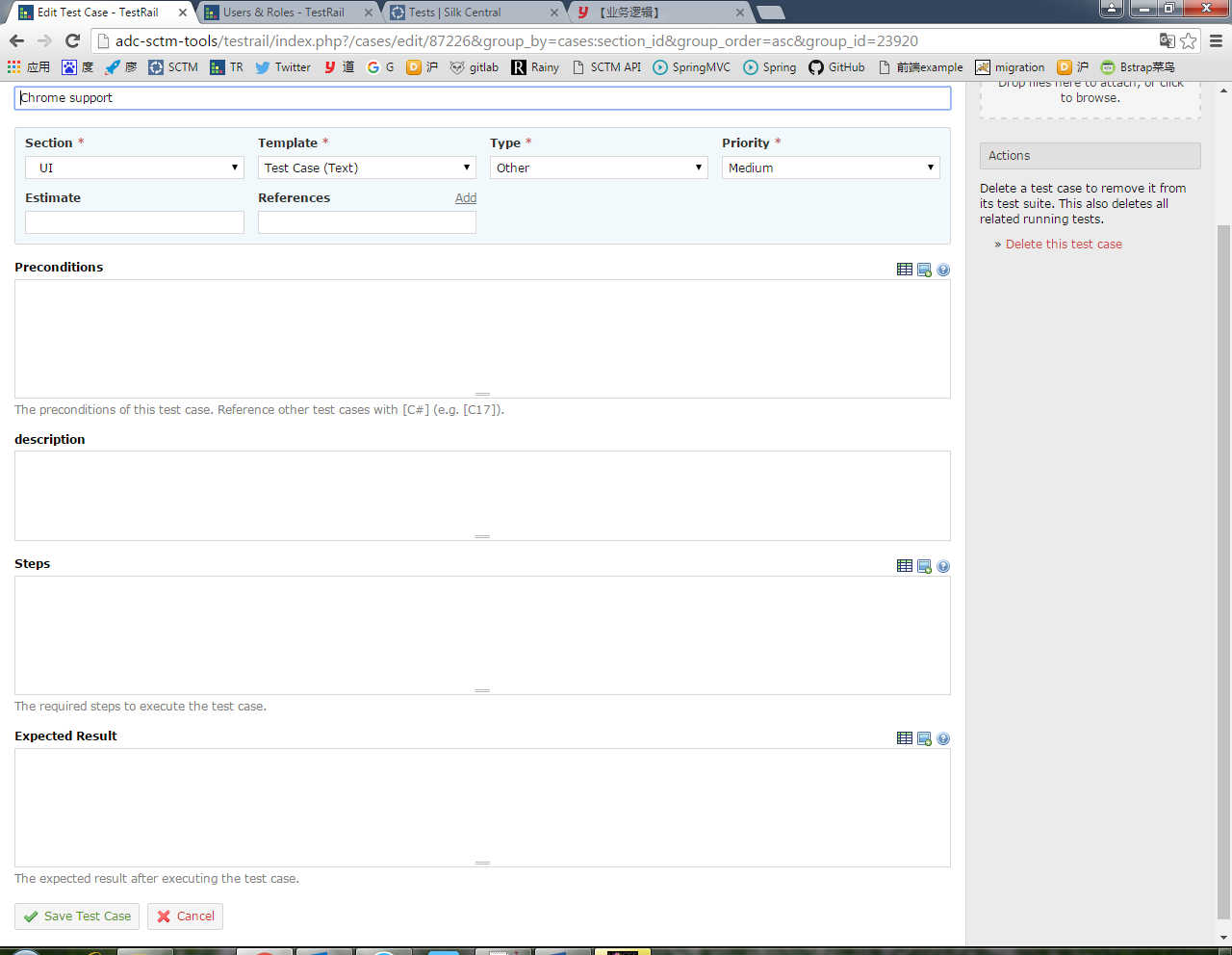
点击version 1.0下的UI下的Add Case按钮。



需要我们填写Case标题。我们写入“Chrome support”后点击绿色的小勾。用同样的方法写入“IE support”、“FireFox support”。

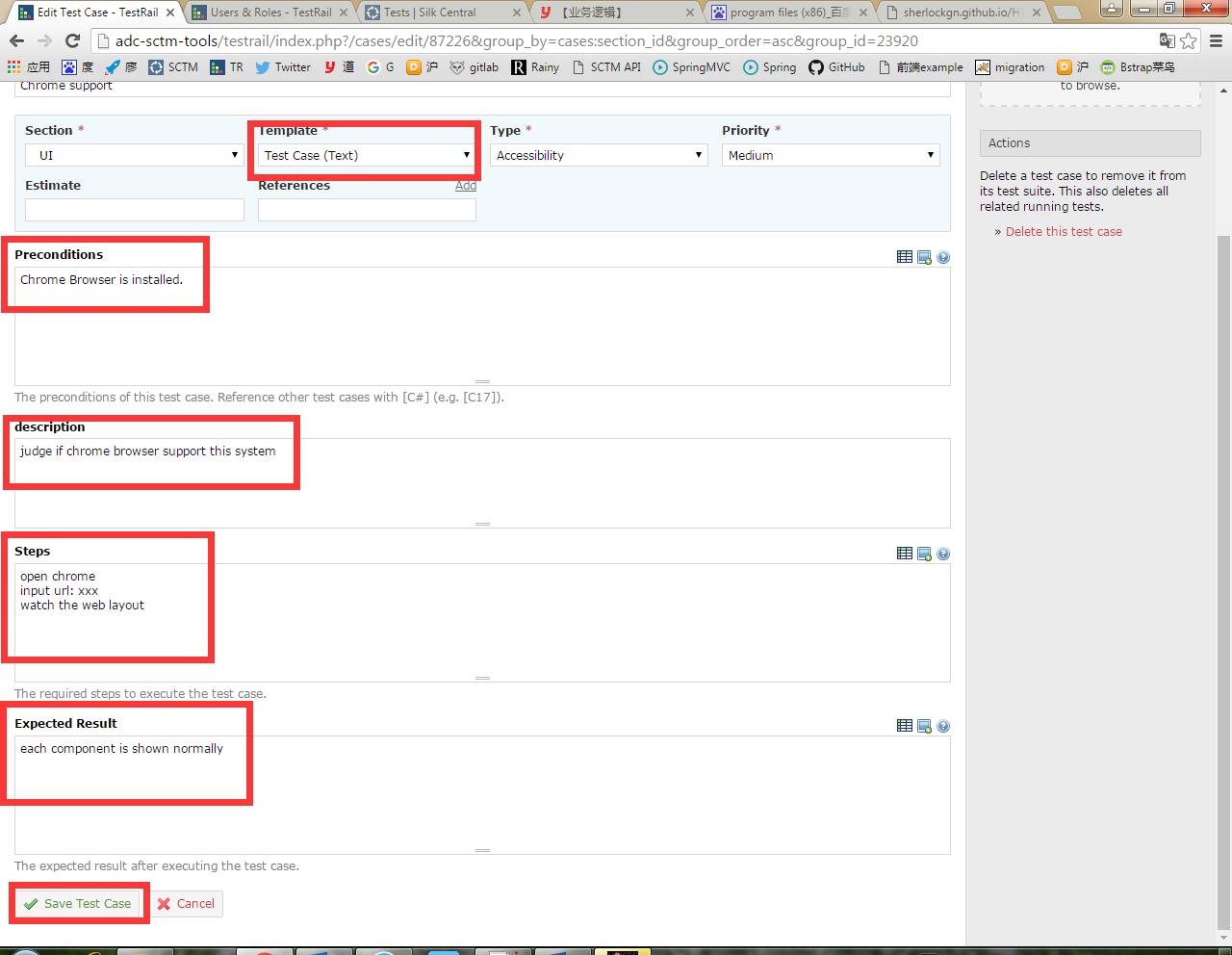


下面我们需要对case进行详细配置。点击“Chrome support”后面的铅笔符号进入详细配置页面。

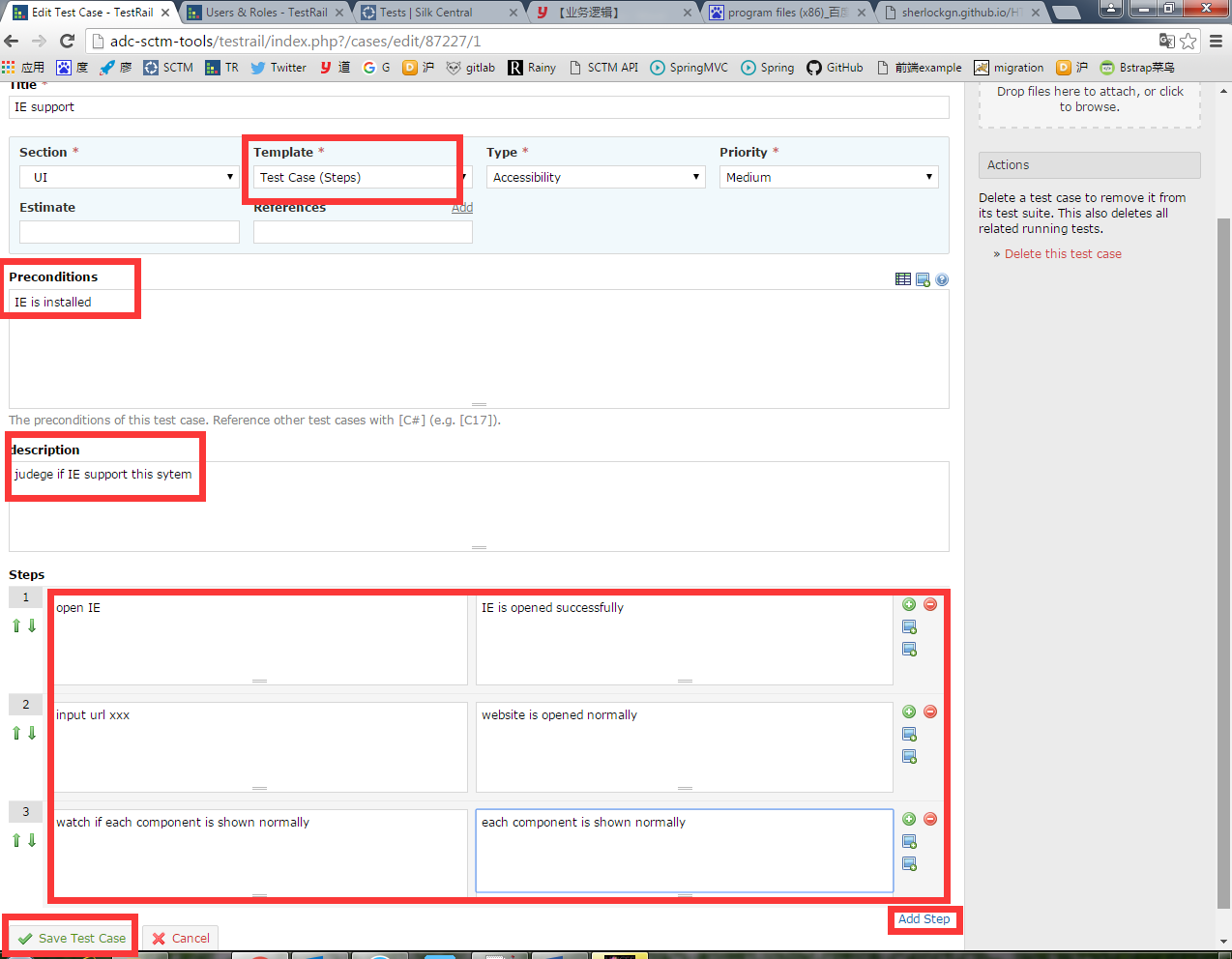


根据模版的不同，我们有不同的case设计方式。

在Chrome support中我们采用Text形式的template：



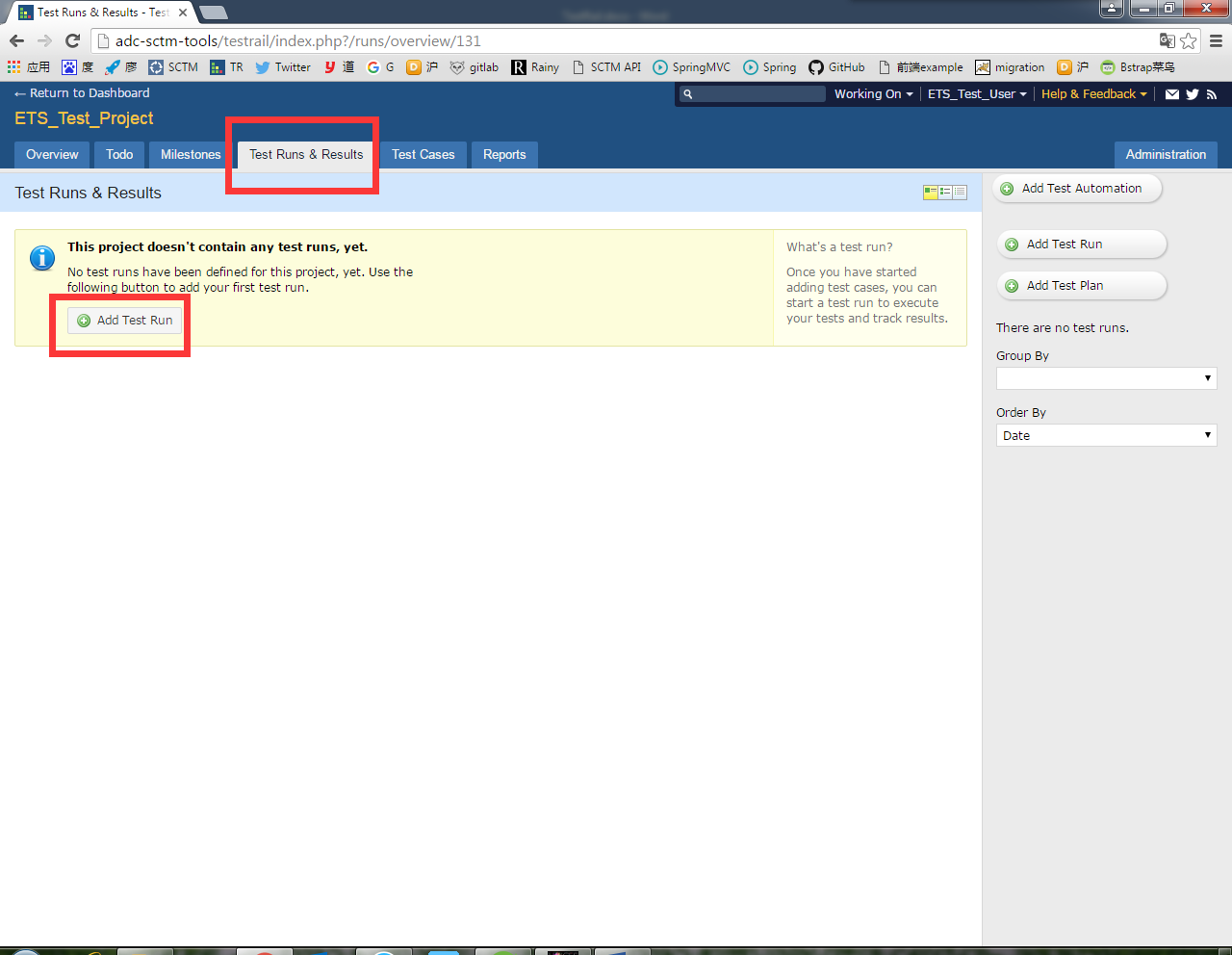
在IE support中我们采用Step形式的template



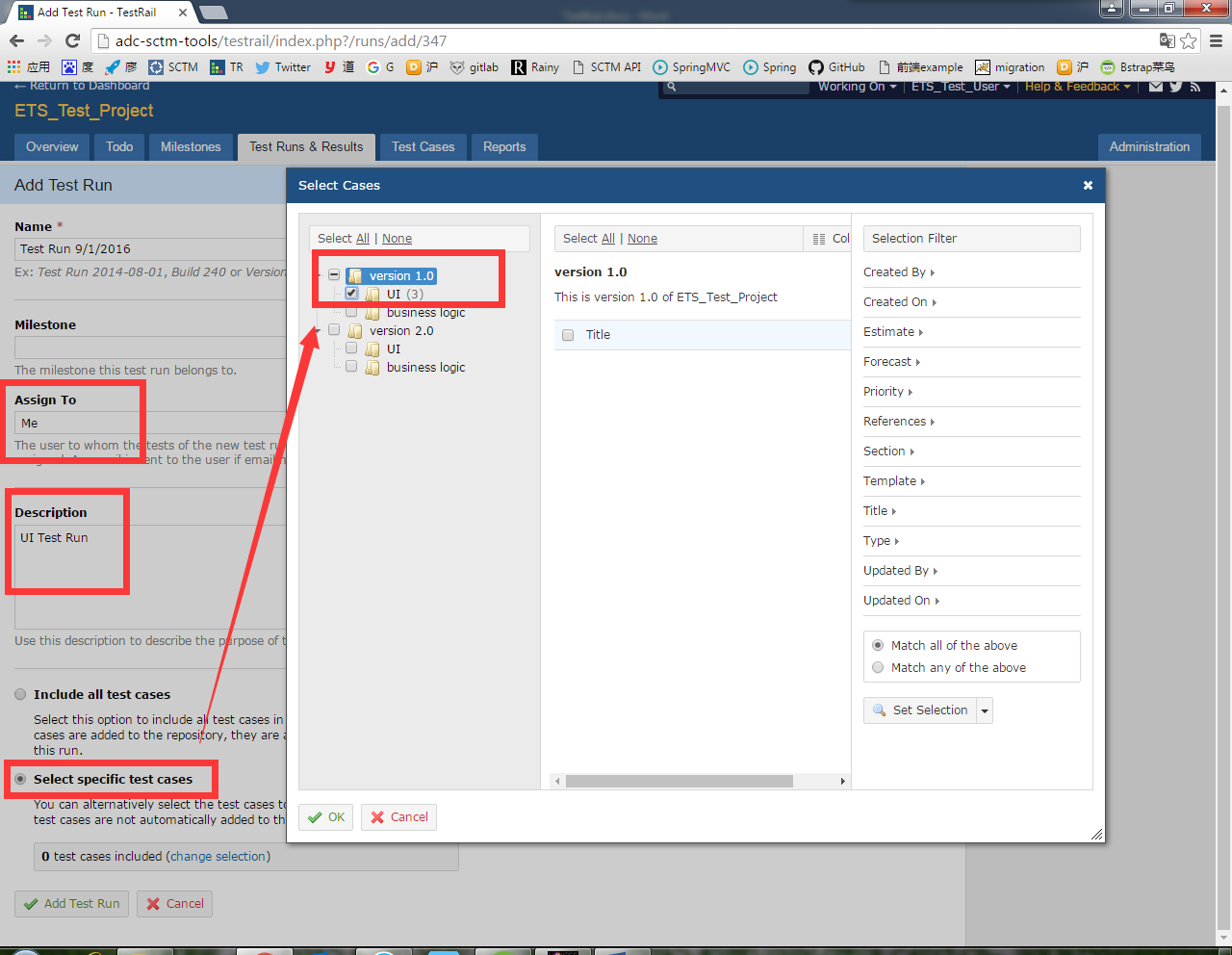
模版Exploratory Session和Text类似，只不过去掉了Preconditions、Steps、Expected Result，增加了Mission、Goals两个输入框。这里不再演示。

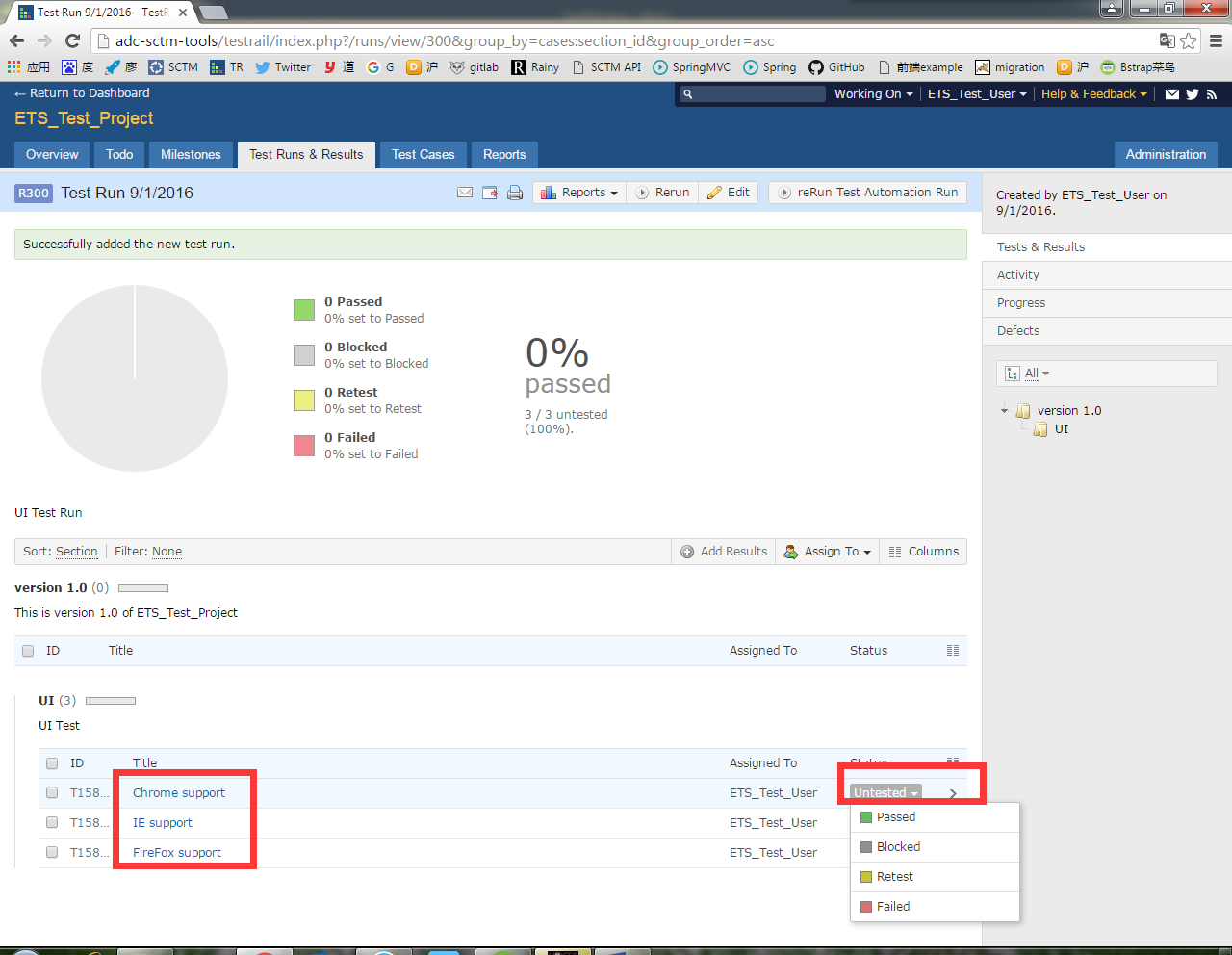
测试用例的执行

点击Test Runs and Results标签，点击Add Test Run按钮添加用例执行。

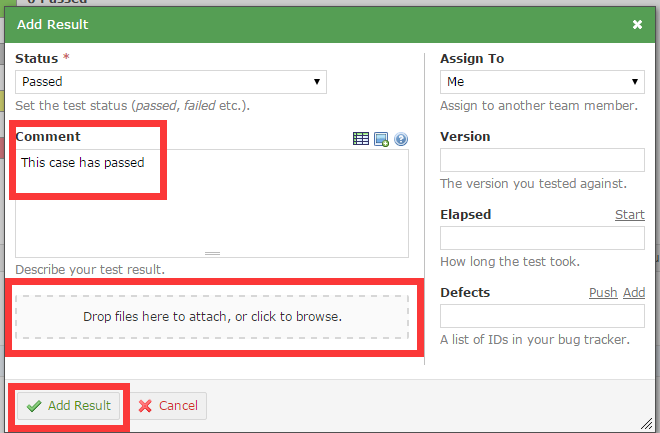


点击完成后，在新的页面中填写分配的组员、描述。测试用例的执行可以是所有项目中的用例或者是指定的用例，如图所示，图中选定了刚刚建立的三个用例。



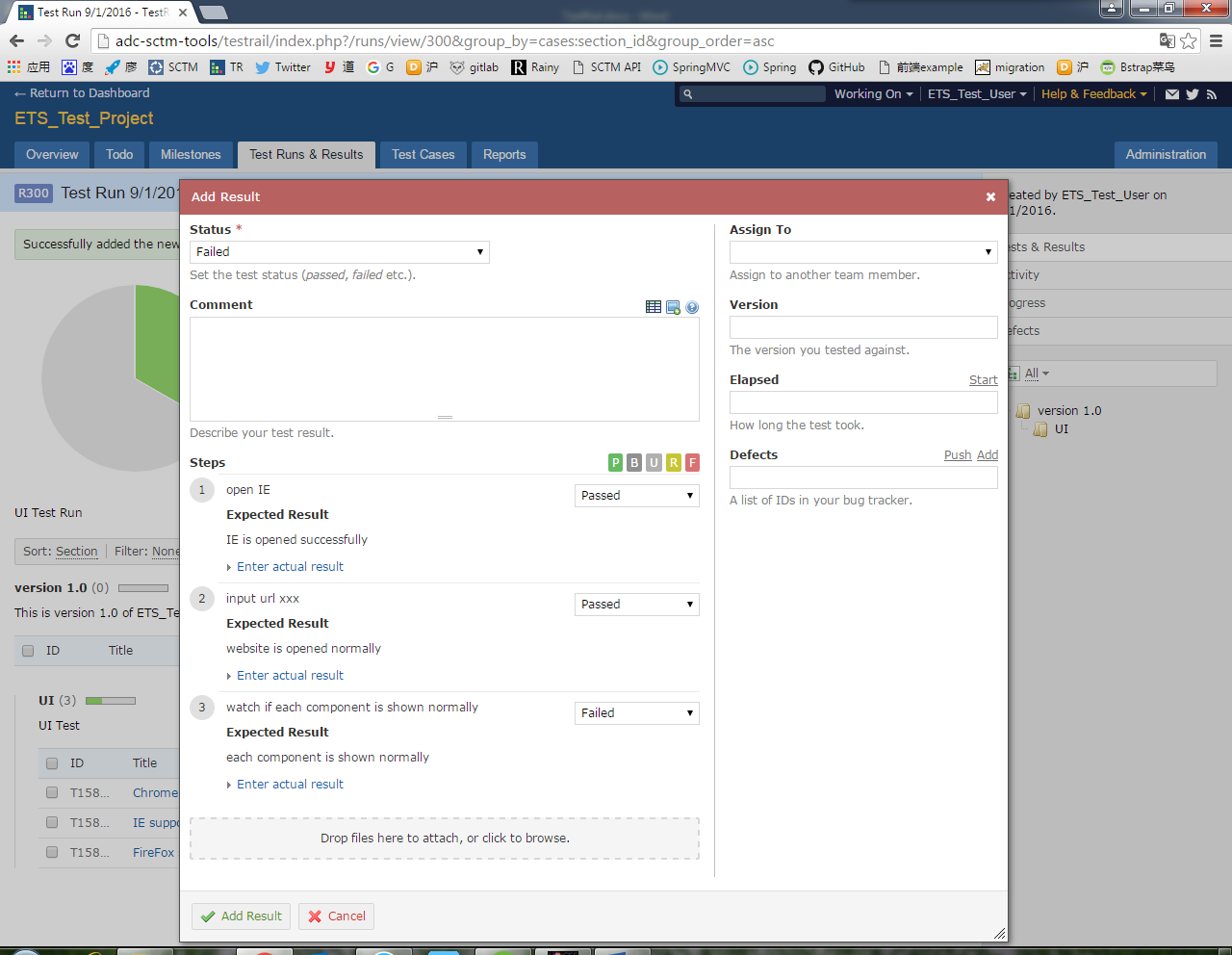


为第一个用例选择Passed，会弹出一个窗口。



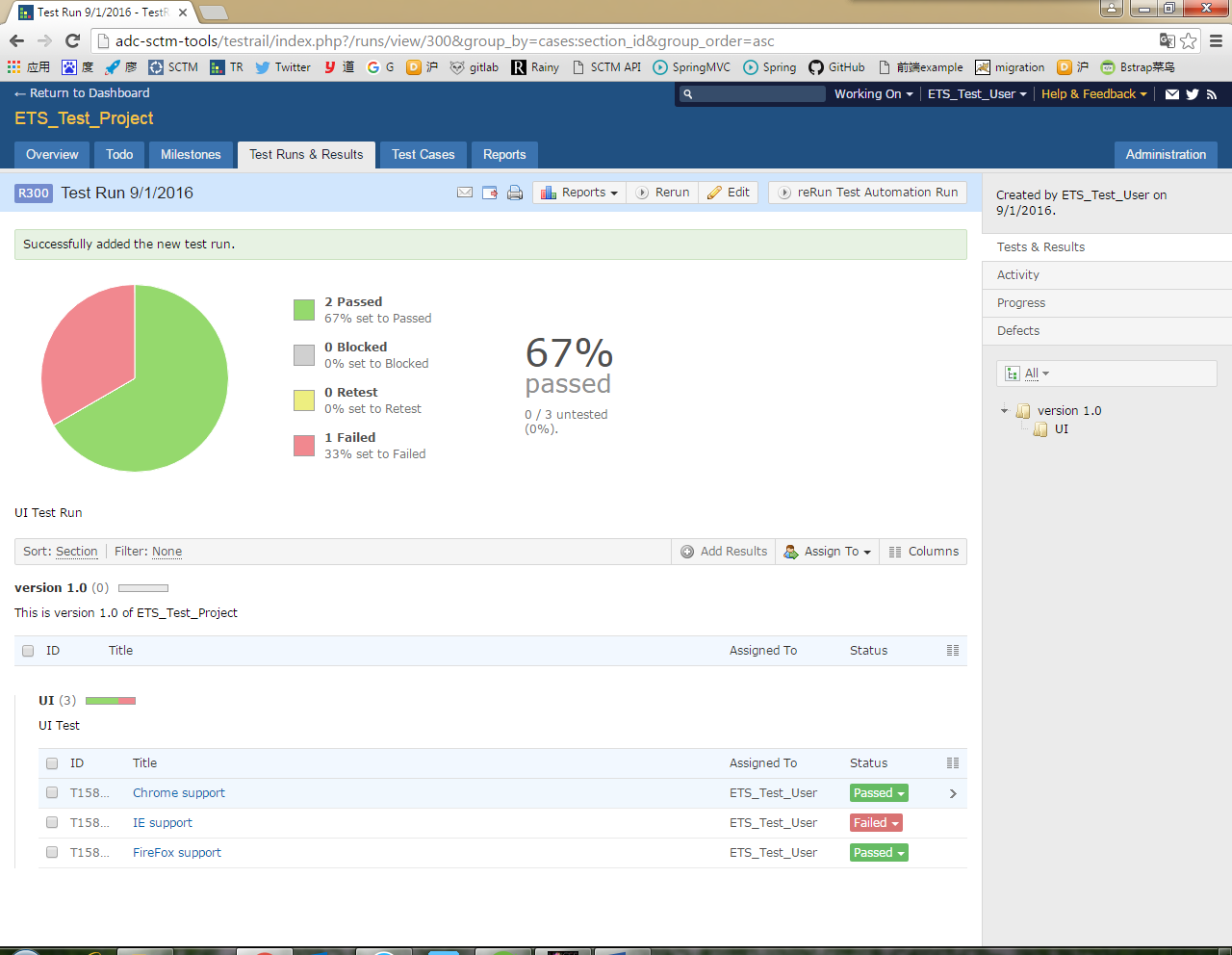
在这个窗口中填写必要的说明后，点击Add Result按钮，则成功设置为Passed。

如果用例采用了step模版，则需要为每个step设置运行结果。

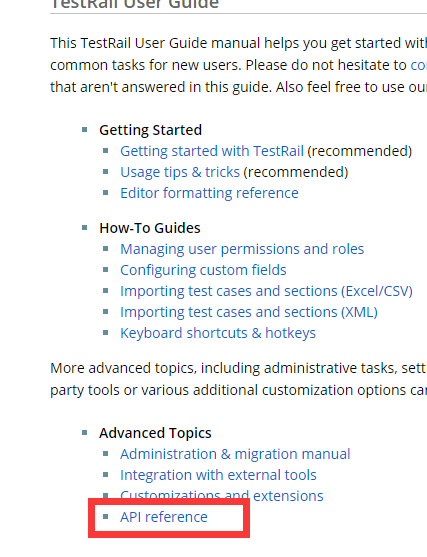
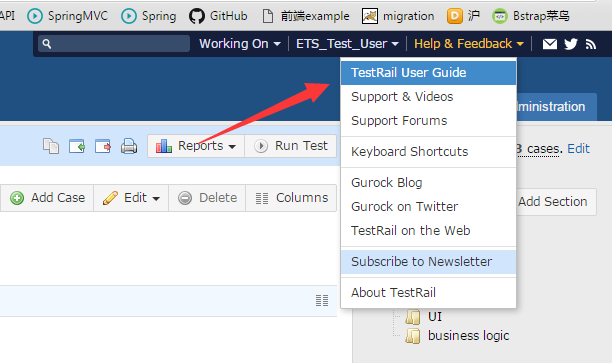


用同样的方法，我们为其他用例设置测试结果，预置的测试结果有Passed、Blocked

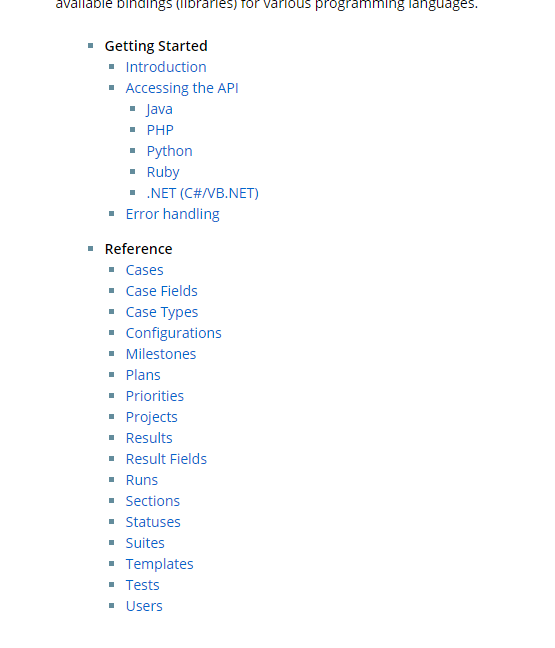
、Retest、Failed四种，可以在Administrator中进行添加与修改。



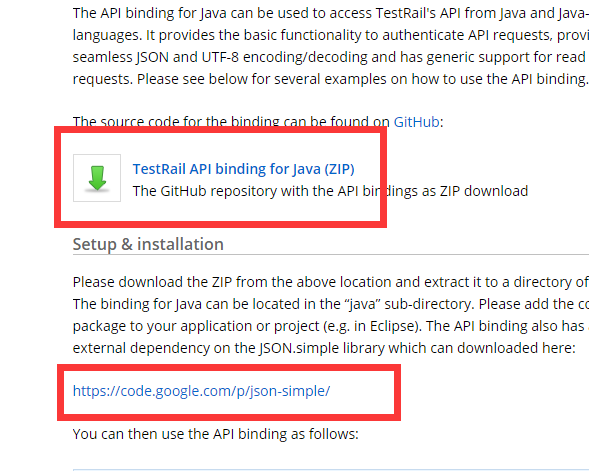
调用TestRail API实现自动测试：



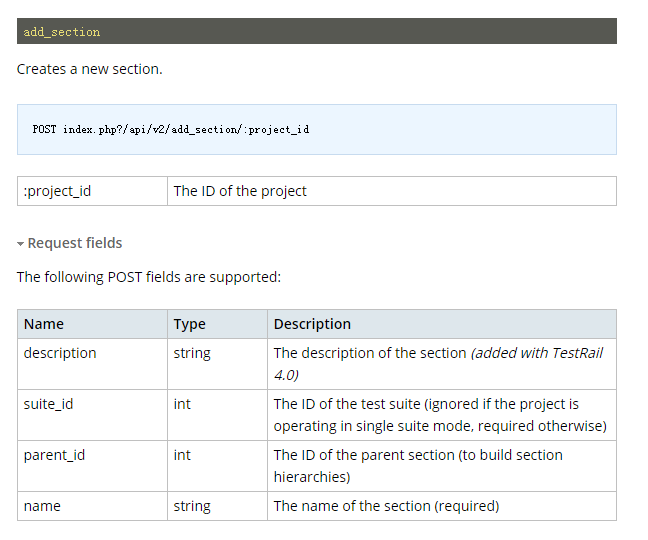
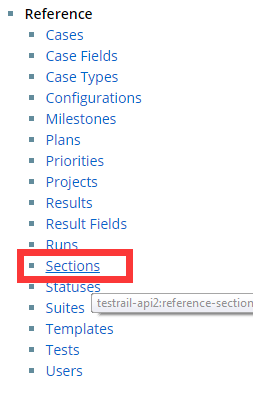
点击Help & Feedback菜单下的TestRail User Guide选项，在弹出的新窗口中点击API reference，即可访问TestRail API。



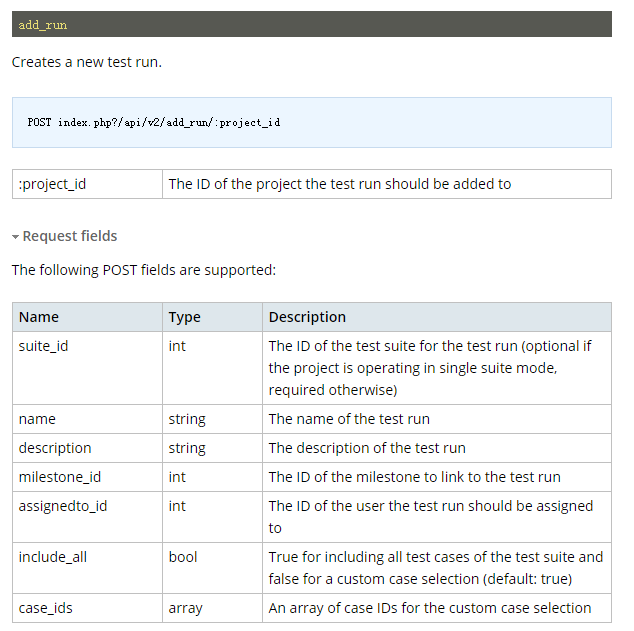
我们以java为例，讲解API的用法，为了正常使用API，我们需要下载依赖包TestRail API以及Json相关包。

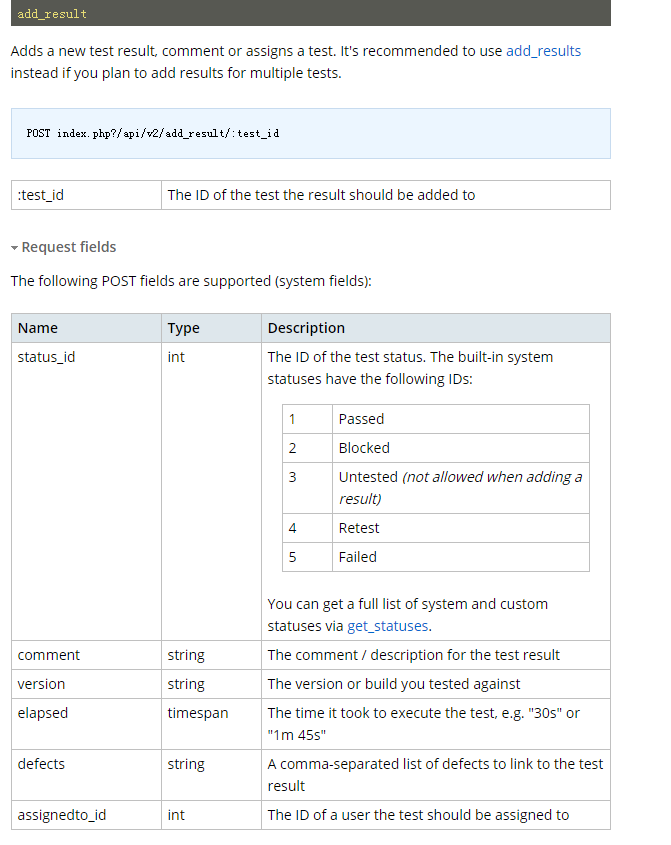


例如我们需要创建section，我们点击Sections链接，在弹出的新页面中找到add\_section接口，即可了解应该传送什么样的参数。



同样我们可以查询如何添加run以及result





下面我们进行一个简单的实例，在刚刚的ETS\_Test\_Project下建立一个独立的section，并在下面添加10个case，用于测试一个加法函数，加法函数有80%的概率获得正确结果，20%概率获得错误结果。

（1）引入依赖包com.gurock.testrail、org.json.simple、org.json.simple.parser；

（2）建立Step、TestCase、TestCaseIdAndResult模型类；

**public** **class** Step {

**private** String stepContent;

**private** String stepExpected;

**public** Step() {}

**public** Step(String stepContent, String stepExpected) {

**this**.stepContent = stepContent;

**this**.stepExpected = stepExpected;

}

……

**public** **class** TestCase {

**private** String title;

**private** **int** type;

**private** List<Step> stepLst;

**private** **int** result;

**private** String comment;

**final** **public** **static** **int** ***Acceptance*** = 1;

**final** **public** **static** **int** ***Accessibility*** = 2;

**final** **public** **static** **int** ***Automated*** = 3;

**final** **public** **static** **int** ***Compatibility*** = 4;

**final** **public** **static** **int** ***Destructive*** = 5;

**final** **public** **static** **int** ***Functional*** = 6;

**final** **public** **static** **int** ***Other*** = 7;

**final** **public** **static** **int** ***Performance*** = 8;

**final** **public** **static** **int** ***Regression*** = 9;

**final** **public** **static** **int** ***Security*** = 10;

**final** **public** **static** **int** ***Smoke\_Sanity*** = 11;

**final** **public** **static** **int** ***Usability*** = 12;

**final** **public** **static** **int** ***Passed*** = 1;

**final** **public** **static** **int** ***Blocked*** = 2;

**final** **public** **static** **int** ***Untested*** = 3;

**final** **public** **static** **int** ***Retest*** = 4;

**final** **public** **static** **int** ***Failed*** = 5;

……

**public** **class** TestCaseIdAndResult {

**private** Long testCaseId;

**private** String comment;

**private** **int** result;

……

（3）建立Automation类，封装TestRail API方法

其属性有：

**private** **long** projectId;

**private** List<TestCase> testCaseLst;

**private** String sectionName, runTitle;

**private** APIClient client;

**private** Long sectionId;

**private** List<TestCaseIdAndResult> testCaseIds;

**private** Long runId;

**private** **boolean** openlog = **true**;

方法1：初始化APIClient

**public** Automation(String account, String password, **long** projectId,

String sectionName, String runTitle) {

**this**.projectId = projectId;

**this**.sectionName = sectionName;

**this**.runTitle = runTitle;

testCaseLst = **new** ArrayList<TestCase>();

testCaseIds = **new** ArrayList<TestCaseIdAndResult>();

client = **new** APIClient("http://adc-sctm-tools/testrail/");

client.setUser(account);

client.setPassword(password);

}

方法2：在TestRail项目的根节点添加section

**private** **void** createSection() **throws** MalformedURLException, IOException,

APIException {

Map<String, Object> m = **new** HashMap<String, Object>();

m.put("name", sectionName);

JSONObject c = (JSONObject) client.sendPost("add\_section/" + projectId,

m);

sectionId = (Long) c.get("id");

}

方法3：将TestCase对象加入List

**public** **void** addTestCase(String title, **int** type, List<Step> stepLst,

String comment, **int** result) {

TestCase tc = **new** TestCase(title, type, stepLst, comment, result);

testCaseLst.add(tc);

}

**public** **void** addTestCase(TestCase testCase) {

testCaseLst.add(testCase);

}

方法4：将List里面的TestCase对象插入到TestRail中

**private** **void** addTestCase() **throws** MalformedURLException, IOException,

APIException {

Map<String, Object> m = **new** HashMap<String, Object>();

**int** cnt = 0;

**for** (TestCase tc : testCaseLst) {

**if** (openlog)

System.***out***.println("add test case: " + (++cnt) + " / "

+ testCaseLst.size());

m.put("title", tc.getTitle());

m.put("type\_id", tc.getType());

m.put("template\_id", 2);

List<Step> conLst = **new** ArrayList<Step>();

**for** (Step step : tc.getStepLst()) {

conLst.add(step);

}

List<Map<String, Object>> mapLst = **new** ArrayList<Map<String, Object>>();

Step.*setArray*(conLst, mapLst);

m.put("custom\_steps\_separated", mapLst);

JSONObject c = (JSONObject) client.sendPost(

"add\_case/" + sectionId, m);

Long testCaseId = (Long) c.get("id");

testCaseIds.add(**new** TestCaseIdAndResult(testCaseId,

tc.getComment(), tc.getResult()));

}

}

方法5：为TestRail项目添加一个Run

**private** **void** addRun() **throws** MalformedURLException, IOException,

APIException {

Map<String, Object> m = **new** HashMap<String, Object>();

m.put("name", runTitle);

List<Long> lst = **new** ArrayList<Long>();

**for** (TestCaseIdAndResult tciar : testCaseIds)

lst.add(tciar.getTestCaseId());

m.put("case\_ids", lst);

m.put("include\_all", **false**);

JSONObject c = (JSONObject) client.sendPost("add\_run/" + projectId, m);

runId = (Long) c.get("id");

}

方法6：为之前添加的Run添加Result

**private** **void** addResult() **throws** MalformedURLException, IOException,

APIException {

**int** cnt = 0;

**for** (TestCaseIdAndResult tcr : testCaseIds) {

**if** (openlog)

System.***out***.println("add result: " + (++cnt) + " / "

+ testCaseIds.size());

**if** (tcr.getResult() == TestCase.***Untested***)

**continue**;

Map<String, Object> m = **new** HashMap<String, Object>();

m.put("status\_id", tcr.getResult());

m.put("comment", tcr.getComment());

client.sendPost("add\_result\_for\_case/" + runId + "/"

+ tcr.getTestCaseId(), m);

}

}

方法7：设置日志开关

**public** **void** logOn() {

openlog = **true**;

}

**public** **void** logOff() {

openlog = **false**;

}

方法8：依次调用上面已定义的方法

**public** **void** run() **throws** MalformedURLException, IOException, APIException {

**if** (openlog)

System.***out***.println("now creating a section");

createSection();

**if** (openlog)

System.***out***.println("now adding test cases");

addTestCase();

**if** (openlog)

System.***out***.println("now adding run");

addRun();

**if** (openlog)

System.***out***.println("now adding results");

addResult();

**if** (openlog)

System.***out***.println("autumation test over");

}

最后我们使用MainTest类为项目添加测试用例。

1.定义目标测试方法：

**public** **static** **boolean** probability(**int** x) {

**int** value = **new** Random().nextInt(100);

**return** value <= x;

}

**public** **static** **int** add(**int** arg1, **int** arg2) {

**if** (*probability*(80)) **return** arg1 + arg2;

**else** **return** arg1 + arg2 + 1;

}

2.在main方法中进行添加section、run以及result

Automation auto = **new** Automation("g405252865@163.com", "gongtianhe123", 131, "mysection", "myrun");

Random r = **new** Random();

**for** (**int** i = 0; i < 10; i++) {

List<Step> lst = **new** ArrayList<Step>();

**int** arg1 = r.nextInt(1000);

lst.add(**new** Step("generate arg1: " + arg1, "arg1 " + arg1

+ " is generated"));

**int** arg2 = r.nextInt(1000);

lst.add(**new** Step("generate arg2: " + arg2, "arg2 " + arg2

+ " is generated"));

**int** funresult = *add*(arg1, arg2);

lst.add(**new** Step("use function add", funresult + " is calculated"));

**int** addresult = arg1 + arg2;

lst.add(**new** Step("use ordinary add", addresult + " is calculated"));

lst.add(**new** Step("compare the 2 numbers", funresult + " == "

+ addresult));

**boolean** s = funresult == addresult ? **true** : **false**;

String success = "we have the correct answer, obviously "

+ funresult + " equlas to " + addresult;

String failure = "that is not correct. function result is "

+ funresult + " but the real add result is " + addresult

+ ", and function is " + (funresult - addresult)

+ " bigger than that of add result";

auto.addTestCase("test case: " + i, TestCase.***Automated***, lst, s ? success : failure, s ? TestCase.***Passed*** : TestCase.***Failed***);

}

**try** {

auto.logOn();

auto.run();

} **catch** (MalformedURLException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **catch** (APIException e) {

e.printStackTrace();

}

运行main方法，得到输出日志。

now creating a section

now adding test cases

add test case: 1 / 10

add test case: 2 / 10

add test case: 3 / 10

add test case: 4 / 10

add test case: 5 / 10

add test case: 6 / 10

add test case: 7 / 10

add test case: 8 / 10

add test case: 9 / 10

add test case: 10 / 10

now adding run

now adding results

add result: 1 / 10

add result: 2 / 10

add result: 3 / 10

add result: 4 / 10

add result: 5 / 10

add result: 6 / 10

add result: 7 / 10

add result: 8 / 10

add result: 9 / 10

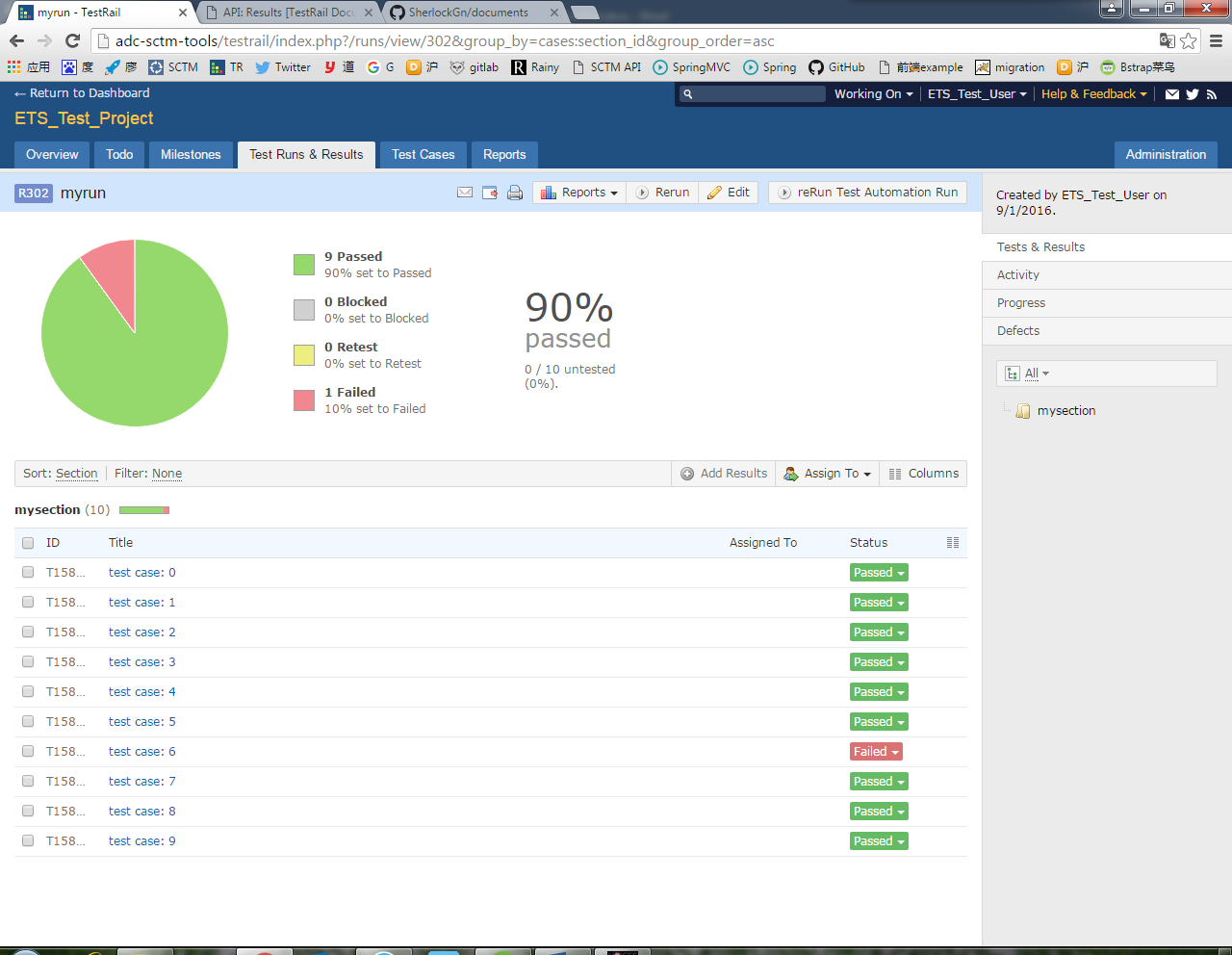
add result: 10 / 10

autumation test over

进入TestRail系统后，我们发现项目下新增了一个section，叫做mysection。



进入Test Run & Result，我们发现多了一个名为myrun的运行记录，里面有10个Result，分别对应之前的10个case。



点击每个result记录，我们可以观察详细结果。

