Web Design Patterns and T2 Components (DRAFT)

Massimo Di Pierro

October 21, 2008

1 Introduction

Source Code:

`http://mdp.cti.depaul.edu/examples/static/web2py.app.plugin_t2.tar`

Video Tutorial (old):

`http://www.vimeo.com/1790354`

In this paper we discuss typical design patterns that seem common to many web applications. We do not claim that all web applications should follow our patterns nor that the patterns described here are in any way unique, but we do believe that the majority of the web applications can be implemented or re-implemented using the patterns described here. The purpose of identifying such patterns is that of defining and building reusable software components and simplifying the development of web applications.

We provide an implementation of all the patterns described here. Our code works on most database engines including Oracle and the Google App Engine.

Here we distinguish the developer of a web application from the user (or visitor) of a web application. We also distinguish three types of patterns: low-level patterns, high-level patterns, and aesthetical patterns. Low-level patterns concern exclusively the developer; the user does not see them and does not need to know about them. High-level patterns instead are perceived by the user and basically describe the building blocks that comprise web pages and how these blocks are logically connected. Aesthetical patterns only deal on the presentation of each of those building blocks and each individual page.

In this paper notes we are mainly interested in high-Level patterns.

Acknowledgements

Bill, CJ, Keith, Krista and Yarko for correcting errors.

All the JS and CSS files used in the example belong to their respective owners.

2 Low-Level Patterns

Low-level patterns concern exclusively the developer, the user does not see them and does not need to know about them. Here we simply provide a list of the most important and recurrent patterns:

- Concurrency: The ability of a web application to serve multiple clients at the same time. Often concurrency is handled by the web server but it does affect the application because of possible race conditions.

- Logging: All requests have to be logged (including their source, path, time stamp and HTTP response number).

- Transactional Safety: The ability of a web application to store and retrieve persistent information (for example in a database) in such a way that information does not get corrupted or becomes inconsistent.

- Database Abstraction: The ability of a web application to communicate with the database in a way that is independent of the type of database engine. For example, there any many SQL dialects and using raw SQL locks the application into one database solution. A Database Abstraction Layer instead provides APIs that write SQL dynamically for different types of database engines.

- Session Management: A session is the data that is uniquely associated to one user and its continuous interaction with the web site. Typically a session stores information about the user, whether the user is authenticated, and in general the state of that user on the web site. Sessions should be stored server side (never sent to the user) and retrieved via the use of secure cookies (that only travel on a Secure Socket Layer), where a cookie only contains a unique identifier that appears random and is unique.

- Request and Response objects: Any web application deals with an HTTP request and generates an HTTP response, which contain data in various forms. For example, there are variables in the HTTP header, some of the variables may contain cookies which may contain other information, etc. There should be an automated way to write and parse that data.

- Exception Catching and Logging: Any web application is going to have bugs, not just in development mode, but also in production mode. All exceptions have to be caught, logged and the user has to be notified without exposing source code or any other information that may compromise the security of the system.

- Model-View-Controller: Most modern web applications are broken in these three components represented by different types of files. The Model files contain a representation of the data stored by the database (loosely speaking the database tables). The View files contain a representation of how data is presented to the user. The Controller files contain a representation of the application logic and business flow. The high-level patterns that we will discuss later are transversal in respect to the MVC, since they require a model, a controller and a view in order to work.

- Validation: All the input that comes from a user has to be validated by the web application, since the user cannot be trusted. This is the most import aspect of web applications security. Examples of validation includes validation of the path in the URL string, validation of the variables in the HTTP header, validation of variables returned via form submission (GET and POST).

- Form Self-Submission (postback): It used to be common to create a `<form>...</form>` that submits the form variables to a different page. This is no longer considered good practice. Forms should submit to the same page that generated the form, perform validation and, on success, redirect the user to a target page. This is because it is cleaner to implement a form as an object which implements self-awareness: the form object knows how to serialize itself in HTML, receive submitted variables, validate them, notify the user if there are errors, communicate with the database in case of successful submission.

- Internationalization: Users visiting the web application coming from different countries and speaking different languages may need to see things in different ways. For example, the text generated by the application (such as error messages) may need to be translated to their language.

- Flash: There should be a single and unified way to notify the user about things that happen e.g. "Your account was created", "You search for something that does not exist", etc. .

- Streaming: Eventually any modern web application needs to upload and download large files. This is done via streaming, i.e. breaking the uploads/downloads into multiple HTTP requests/responses.

- Caching: Some pages may be requested very often and so often that the information they contain does not change from one instance to the other. In this case the application should be able to cache the page and reuse the cached page, in order to minimize the workload. The output of any function should be cachable, should have an expiration time, and it should be possible to force any cache to expire at any time.

- Safe Uploads: When the user uploads a file, this must be dealt with safely. The file must be renamed to prevent directory traversal attacks, stored on the file system or in the database, and linked by the database record. There should also be a mechanism to retrieve and download the file.

- Protocols: There are some common protocols utilized on the web and any modern web application should be able to support them. For example: Atom, RSS, Wiki markup, JSON, XML, CAS, OpenID, etc.

The list above is certainly partial but it provides an example of what we mean by low-level patterns. These are features that concern the implementation of any web application but the user does not need to know how they work. These patterns are so common that they are implemented in various ways and with various degrees of success by many modern web frameworks.

In this paper we will use as reference the WEB2PY framework since it implements all the patterns described above and it makes particularly easy to implement the high-level patterns described in the next section.

3 High-Level Patterns

High-level patterns describe what the user sees from a functional point of view. An example is a page that "allows the user to read and post comments".

In order to catalogue patterns we need to break them into small but general purpose components. We define "small" as something that can be implemented in one single function (or class method). We define "general" as something that can plugged-in into any new or existing application and requires relatively minor customization.

Each of the patterns described below, even if it can be and will be implemented into a single function, will require access to a model (data representation), a controller (application logic that exposes the patterns) and a view (a visual presentation of the pattern).

We have identified the following patterns:

- **Menu**: Display a navigation menu
- **Create**: The user is presented with a Create form for inserting a new record in a database table. All user input should be validated, the form should display error messages (if any), and should only allow the user to edit the values of some fields. Some fields may have default values. A Create form may include file upload fields.
- **Display**: The user is presented with a visual representation of the data in one database record.
- **Update**: The user is presented with an Update form for changing the information in a database record. This is very similar to a Create form except that the form is pre-populated. If the record contains an uploaded media file (such as an image) it should show a preview of the current value and allow its download.

- **Delete**: The user can delete a database record. (The previous four items here are usually referred to as CRUD=Create, Read, Update, and Delete.)
- **Itemize**: The user is presented with a list of database records selected from a table. The list may contain results from one table or a JOIN between multiple tables, based on the query criteria. Data should be paginated if the list is too long.
- **Search**: The user is presented with an interface that allows filtering and sorting of records.
- **Attachments**: A record may have attachments, that is files that are uploaded and are associated to the record. The user may submit a new attachment or delete existing ones. It should be possible to preview and download attachments.
- **Comments**: A record may have comments, that is statements posted by users about the record. The user can read previous comments and post a new comment. Comments are organized in a tree-like structure.
- **Reviews**: A record may have reviews. These are like comments but they are organized in sequential order (a review is not a response to another review, while a comment may be) and each comment also includes an integer numerical rating (often visualized with stars).
- **Login**: The user should be able to login into the web application.
- **Logout**: The user should be able to logout from the web application.
- **Require Login**: Some pages require that the user is logged in.
- **Group Management**: Users are normally organized in Groups. The same user can be member of multiple groups and groups can contain many members. Group membership can be automatic, automatic upon request, or requiring authorization. There may be different types of membership. Group membership should be determined at login and eventually updated as required.
- **Access Management**: Access to resources is granted on a per-group basis. A user can access a given resource only if the user belongs to a

group that has access to the object. In general there are different types of group membership and different types of accesses. Access should be checked before any CRUD operation.

- **File Download**: Implements file download.
- **Stamp**: Automatically Stamp table records for creation date, author, last modification date, and author of the last modification.
- **On Error Redirection**: If some error occurs due to invalid user input, other than invalid form submission, redirect the user to an error page.

We have implemented each one of this patterns in a WEB2PY plugin called “plugin_t2” (T2 stands for WEB2PY tier two). For “tier” we adopt this definition: ”a single layer of packages forming part of a unit load.”

In this context a plugin is set of modules, models, views, controllers and static files that can be shared by multiple applications.

In particular we created a python module called `t2.py` that defines a class called `T2` and one instance of that class called `t2`. The methods implementing the patterns described can be accessed via the `t2` instance.

In the next chapter we discuss them one by one and provide examples of usage but here we provide a simple example. Let’s consider the following minimalist web2py application that is comprised of the following files:

- `models/db.py`

```
1 db=SQLDB('sqlite://storage.db')
2 db.define_table('puppy',
3     SQLField('name'),
4     SQLField('image','upload'))
5 db.puppy.name.requires=IS_NOT_EMPTY()
6 db.puppy.represent=lambda row: A(row.name,_href=t2.action('
     display_puppy',[row.id]))
7
8 from applications.plugin_t2.modules.t2 import T2
9 t2=T2(request,response,session,cache,T,db)
```

  Here line 5 sets a validator and lines 6-7 set a representation for a puppy record. It displays the puppy by name with a link to a ”display_puppy” action.

- `controllers/default.py`

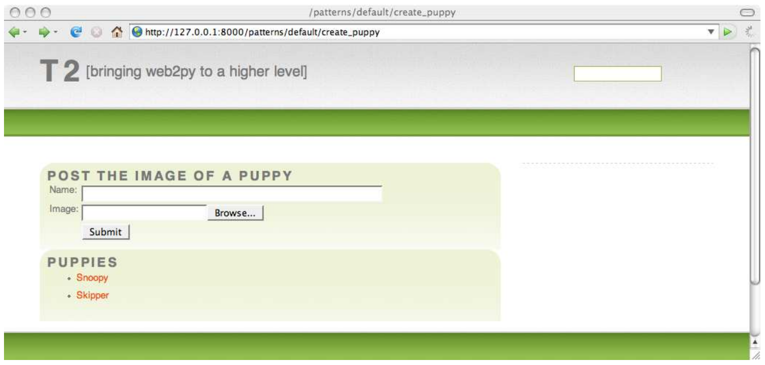
```
def create_puppy():
    form=t2.create(db.puppy)
    itemize=t2.itemize(db.puppy)
    return dict(form=form,itemize=itemize)
```

- `views/layout.htm` (the provided layout, unmodified)
- `views/default/create_puppy.html` (the view for the index action)

```
{{extend 'layout.html'}}
<div class="frame">
  <h2>Post the image of a Puppy</h2>
  {{=form}}
</div>
<div class="frame">
  <h2>Puppies</h2>
  {{=itemize}}
</div>
```

- The provided static files required by the default layout.

This complete program generates the following fully functional web application that allows users to post images of puppies.


/patterns/default/create_puppy
http://127.0.0.1:8000/patterns/default/create_puppy
T2 [bringing web2py to a higher level]
POST THE IMAGE OF A PUPPY
Name:
Image:
Browse...
Submit
PUPPIES
Snoopy
Skipper

The program handles file uploads, form validation and user notification. Let us now add two new controller actions to `default.py`

```
def create_puppy():
    form=t2.create(db.puppy)
    itemize=t2.itemize(db.puppy)
    return dict(form=form,itemize=itemize)

```

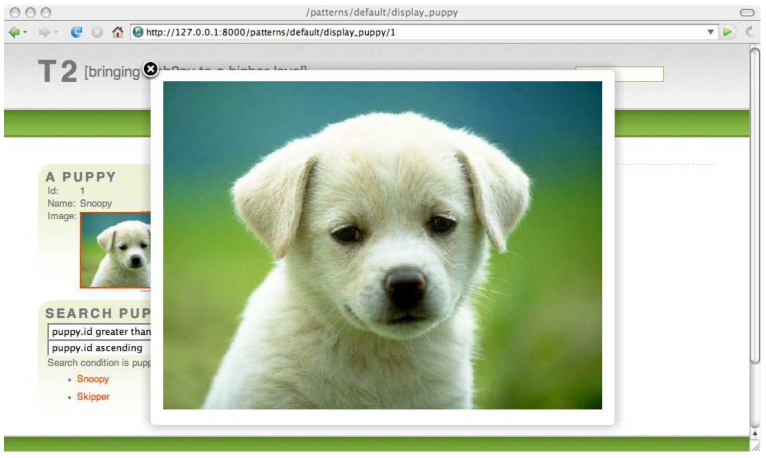
```
def display_puppy():
    puppy=t2.display(db.puppy)
    search=t2.search(db.puppy)
    return dict(puppy=puppy,search=search)

def download(): return t2.download()
```

And an a view `views/default/display_puppy.html` associated to the new view:

```
{{extend 'layout.html'}}
<div class="frame">
  <h2>A Puppy</h2>
  {{=puppy}}
</div>
<div class="frame">
  <h2>Search Puppies</h2>
  {{=search}}
</div>
```

Notice how `display` figures out which puppy to display from the record_id in the command line argument (REST). The `t2.display` function does not just show the puppy, it also creates an image preview and the user can zoom-in by clicking on it. The `t2.search` function creates a search widget that can be used to search puppies by id, name, and image name (the only fields that are exposed).

Here is a screen shot:


/patterns/default/display_puppy
http://127.0.0.1:8000/patterns/default/display_puppy/1
T2 [bringing
A PUPPY
SEARCH PUP
puppy.id greater than
puppy.id ascending

Id: 1
Name: Snoopy
Image:

Search condition is pupp

- Snoopy
- Skipper

Now we can easily add authentication to our application by modifying the `default.py` controller as follows:

```
 1 response.menu=[
 2 ['puppies',False,t2.action('create_puppy')],
 3 ['login',False,t2.action('login')],
 4 ['logout',False,t2.action('logout')],
 5 ['register',False,t2.action('register')]
 6 ]
 7 
 8 def register(): return dict(form=t2.register())
 9 def login(): return dict(form=t2.login())
10 def logout(): t2.logout(next='login')
11 def index(): t2.redirect('create_puppy')
12 
13 @t2.requires_login('login')
14 def create_puppy():
15     form=t2.create(db.puppy)
16     itemize=t2.itemize(db.puppy)
17     return dict(form=form,itemize=itemize)
18 
19 @t2.requires_login('login')
20 def display_puppy():
21     puppy=t2.display(db.puppy)
22     search=t2.search(db.puppy)
23     return dict(puppy=puppy,search=search)
24 
25 @t2.requires_login('login')
26 def download(): return t2.download()
```

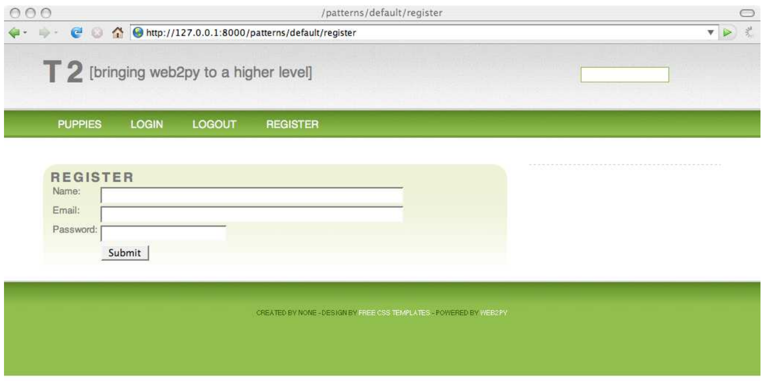
where the first 6 lines, create a menu, lines 8-11 create the registration pages. For the latter to work properly we need to implement the following views for `login` for `login`

```
{{extend 'layout.html'}}
<div class="frame">
  <h2>Login</h2>
  {{=form}}
</div>
```

and for `register`

```
{{extend 'layout.html'}}
<div class="frame">
  <h2>Register</h2>
  {{=form}}
</div>
```

Here is how the registration page looks like:


/patterns/default/register
http://127.0.0.1:8000/patterns/default/register
T2 [bringing web2py to a higher level]
PUPPIES
LOGIN
LOGOUT
REGISTER
REGISTER
Name:
Email:
Password:
Submit
CREATED BY NONE - DESIGN BY FREE CSS TEMPLATES - POWERED BY WEB2PY

Following the same principles, adding

```
{{=t2.comments(db.puppy)}}
```

to the `display_puppy` view will allow visitors to post comments and adding

```
{{=t2.attachments(db.puppy)}}
```

will allow users to post attachment (for example additional pictures).

The value of the `next` argument is used to specify the action where to go upon success. The absolute path to any given action can be generated using the shortcut `t2.action('index')`.

`t2.update`, `t2.delete`, `t2.itemize` can also take additional parameters for example to specify a query. `t2.itemize` can deal with joins. Both `t2.create` and `t2.update` can have callbacks that are called after accepting a user submission.

All required tables are generated as needed. The user can also define tables himself.

4 A More Complex Example

Here is an example of `db.py` that re-defines `t2_person`, defines a puppy table, and defines a relation of friendship between a person and a puppy.

```
db=SQLDB("sqlite://storage.db")

db.define_table('t2_person',
   SQLField('name',requires=IS_NOT_EMPTY()),
   SQLField('password','password',requires=CRYPT()),
   SQLField('email',requires=IS_EMAIL()),
   SQLField('registration_key'))

db.define_table('puppy',
   SQLField('name'),
   SQLField('image','upload'))

db.define_table('friendship',
   SQLField('person_id',db.t2_person),
   SQLField('puppy_id',db.puppy))

friendship=db.t2_person.id==db.friendship.person_id \
       and db.puppy.id==db.friendship.puppy_id

db.t2_person.represent=lambda person:\
    person.name+' is friend of '
db.puppy.represent=lambda puppy:\
    A(puppy.name,_href=t2.action('display',puppy.id))

from applications.plugin_t2.modules.t2 import T2
t2=T2(request,response,session,cache,T,db)
```

The following controller implements authentication (registration, login, logout, login requirements), authorization (only the poster person can edit a puppy info but everybody logged-in can comment and review the puppy) and automatically associates as friend every person with the puppies he/she posts. Persons and Puppies have a many-to-many relation and this reflects in the search widget of the `index` action.

```
if t2.logged_in: response.menu=[
  ['index',False,t2.action('index')],
  ['logout',False,t2.action('logout')]]
else: response.menu=[
  ['login',False,t2.action('login')],
  ['register',False,t2.action('register')]]

def register(): return dict(form=t2.register())
def login(): return dict(form=t2.login())
def logout(): t2.logout(next='login')

@t2.requires_login(next='login')
def download(): return t2.download()


@t2.requires_login(next='login')
```

```
def index():
    form=t2.create(db.puppy,callback=lambda form:\
         t2.add_access(db.puppy,form.vars.id) and \
         db.friendship.insert(person_id=t2.person_id,
                              puppy_id=form.vars.id))
    search=t2.search(db.t2_person,db.puppy,query=friendship)
    return dict(form=form,search=search)

@t2.requires_login(next='login')
def display():
    display=t2.display(db.puppy)
    comments=t2.comments(db.puppy)
    reviews=t2.reviews(db.puppy)
    return dict(display=display,comments=comments,reviews=
       reviews)

@t2.requires_login(next='login')
def update():
    if not t2.have_access(db.puppy,t2.id):
        t2.redirect('index',flash='No access')
    update=t2.update(db.puppy,next='index')
    return dict(update=update)
```

5 API

5.1 State Variables

T2 defines the following state variables:

- `t2.person_id` the id of the person logged in, or None
- `t2.person_name` the name/alias of the person logged in, or None
- `t2.person_email` the email address of the person logged in, or None
- `t2.error_action` the action to call in case of error, defaults to 'error'
- `t2.now` the current date-time
- `t2.request` the web2py request object
- `t2.response` the web2py response object

- `t2.session` the web2py session object
- `t2.cache` the web2py cache object
- `t2.T=T` the web2py translation object
- `t2.db` the database to be used by T2
- `t2.all_in_db` True is binary data and sessions go in DB
- `t2.is_gae` True is running on GAE
- `t2.id` the value of `request.args[-1]` if `request.args` else 0
- `t2.logged_in` True is the person is logged in else False
- `t2.my_groups_id` list of record id's of those groups the person is member of

5.2 Text Messages

T2 defines the following messages that the user can redefine:

- `t2.messages.record_created="Record Created"`
- `t2.messages.record_modified="Record Modified"`
- `t2.messages.record_deleted="Record(s) Deleted"`
- `t2.messages.record_was_altered= "Record Could Not Be Saved Because It Has Changed"`
- `t2.messages.invalid_value="Invalid Entry"`
- `t2.messages.attachment_posted="Attachment Posted"`
- `t2.messages.no_comments="No Comments"`
- `t2.messages.no_visible_comments="No Visible Comments"`
- `t2.messages.review_posted="Review Posted"`
- `t2.messages.register_email_body="...  %(registration_key)s ..."`

- `t2.messages.register_email_subject="Verify Registration"`
- `t2.messages.logged_in="Logged In"`
- `t2.messages.invalid_login="Invalid Login"`
- `t2.messages.logged_out="Logged Out"`

5.3 Methods

T2 defines the following methods.

5.3.1 The constructor

```
T2.__init__(self,request,response,session,cache,T,db,
   all_in_db=False)
```

Use in at the bottom of your model as follows

```
from applications.plugin_t2.modules.t2 import T2
t2=T2(request,response,session,cache,T,db)
```

5.4 Authentication

T2 defines the following authentication methods

```
t2.register(self,verification=False,sender='',next='login',
    onaccept=None)
t2.login(self,next='index',onlogin=None)
t2.logout(self,next='index')
t2.verify(self,next='login')
t2.requires_login(self,next='login')
```

They should be used to define corresponding authentication actions in your controller:

```
def register(): return dict(form=t2.register())
def login(): return dict(form=t2.login())
def logout(): t2.logout()
def verify(): t2.verify()

def index(): return dict()

@t2.requires_login()
def private_page(): return dict()
```

All actions tagged with the `@requires_login()` decorator are not public unless the person is logged in.

By default registration does not perform verification.
If `verification=True` then the registrant is sent an notification email and registration is not completed until the registrant visits the `verify?key=KEY` action. KEY is in the email sent to registrant.

5.4.1 Utility Methods

They are

```
t2.action(self,f=None,args=[],vars={})
t2.redirect(self,f=None,args=[],vars={},flash=None)
```

`t2.action('index')` is a shortcut for

```
URL(r=request,f='index')
```

`t2.redirect('index')` is a shortcut for

```
redirect(URL(r=request,f='index'))
```

5.4.2 View Helpers

They are

```
t2.include(self)
t2.menu(self,menu,style='h')
```

`t2.include` generates the part of the head of the HTML page which includes required CSS and JS files (as listed in `response.files`), while

```
{{=t2.menu(menu)}}
```

generates a dropdown menu. Here is a minimalist T2 sample layout:

```
<html><head>
<title>{{=response.title or URL(r=request)}}</title>
<meta http-equiv="content-type" content="text/html; charset=utf
   -8" />
<meta name="keywords" content="{{=response.keywords}}" />
<meta name="description" content="{{=response.description}}" />
<link href="/{{=request.application}}/static/t2/styles/style.css
   "
     rel="stylesheet" type="text/css" charset="utf-8" />
{{=t2.include()}}
</head><body>
```

```
<div class="menu">{{=t2.menu(response.menu)}}</div>
{{if response.flash:}}
  <div id="flash">{{=response.flash}}<div>
{{pass}}
{{include}}
</body>
```

5.4.3 Download Control

```
t2.download(self)
```

Use it to define an action like in the following example:

```
def download(): return t2.download()
```

It will serve/stream files whose name is in `request.args[0]` Works even if the data is stored in the database.

5.4.4 CRUD Controls

They are

```
t2.create(self,table,next=None,vars={},onaccept=None)
t2.update(self,table,query=None,next=None,
  deletable=True,vars={},onaccept=None,ondelete=None)
t2.delete(self,table,query=None,next=None)
t2.read(self,table,query=None,
  limitby=None,orderby=None)
t2.display(self,table,query=None)
```

To create a self processing create form just define

```
def create_form():
    return dict(form=t2.create(db.mytable))
```

or directly in the view

```
{{=t2.create(db.mytable)}}
```

similarly for `update` and `display` controls. In general the first argument is the table object the control acts on. For `update`, `delete`, `read` and `display` the second argument is an optional web2py query to identify the record(s) involved. If there is no query T2 picks the record with id equal to t2.id which is parsed from the last argument of the URL.

The `next` argument is the name of action where to jump after the form is accepted. If `next` is None, the same page is rendered again.

For `create` and `update`, `vars` is a dictionary of optional variables that need to be passed to inserted in the form.vars before accepting; `onaccept` and `ondelete` are optional callback functions called if the form is accepted and if the form successfully deleted a record. These callback functions should take a single argument, the form object itself.

`Update` also prevents concurrent updates. If two visitors try to update the same record, the first to save will succeed. The second will get an error "Record Could Not Be Saved Because It Has Changed"

5.4.5 List and Search Records

```
t2.itemize(self,*tables,**opts)
t2.search(self,*tables,**opts)
```

Both of them take a list of tables and some optional named arguments. For example:

```
{{=t2.itemize(db.mytable,query=db.mytable.id>0,orderby=db.
    mytable.id)}}
```

displays all records that match the query. If no `limitby` is specified, this control paginates the items, 25 per page.

```
{{=t2.search(db.mytable)}}
```

creates a search widget and called itemize internally. One can optionally specify a query as additional required condition.

If more than one table are specified, they perform an INNER JOIN.

The items are represented, by default, by the first field in the table. This can be changed by

```
db.mytable.represent=lamnda row: '%s %s' % (row.myfield,row.id)
```

etc. One can also use helpers to change the representation of a row.

5.4.6 Additional Controls

```
t2.attachments(self,table,readable=True,writable=True,
  deletable=False)
t2.comments(self,table,moderated=False,readable=True,
  writable=True,deletable=False,hide=True)
t2.reviews(self,table,status='approved',readable=True,
  writable=True,deletable=False,subtitle="You Review")
```

5.4.7 Membership Controls

```
t2.add_membership(self,person_id,group_id,membership_type='
    default')
t2.del_membership(self,person_id,group_id,membership_type='
    default')
t2.have_membership(self,group_id,membership_type='default')
t2.my_memberships(self)
```

add_membership gives the person identified by person_id membership of the group identified by group_id. Optionally one can specify a membershi_type.

del_membership deletes the membership.

have_membership checks if the logged in person has membership if the group.

my_memberships returns a list of membership records for the person logged in.

When a new person registers, a new group is automatically created with the same name as that person and the person is made 'default' member of that group.

5.4.8 Access Controls

Persons have memberships of groups. Groups have access to resources. A resource could be a table or a table record. The controls are:

```
t2.add_access(self,table,record_id=0,access_type='default',
    group_id=None)
t2.del_access(self,table,record_id=0,access_type='default',
    group_id=None)
t2.have_access(self,table,record_id=0,access_type='default')
```

add_access gives access to a resource to the group identified by group_id. If record_id==0 the resource is the entire table, else the resource is the record in the table identified by record_id. If no group_id is specified T2 assumes the group is the one uniquely associated with person logged in. Optionally one can specify different types of access: 'read', 'write', etc.

del_access removes the access.

have_access checks if the logged in user belongs to a group (despite membership type) that has access_type access to the resource (table,record_id).

For example here is an example of code that allows only the person who posts a document to update the document (assuming there is a table called "document").

```
@t2.requires_login()
def post_document():
    form=t2.create(db.document,
      onaccept=lambda form: t2.add_access(db.document,
                                         form.vars.id))
    return dict(form=form)

@t2.requires_login()
def edit_document():
    if not t2.have_access(db.document,t2.id):
        t2.error()
    form=t2.update(db.document)
    return dict(form=form)
```

5.4.9 Email Control

```
t2.email(self,sender,to,subject='test',message='test')
```

Can be used to send emails. Works on GAE too.

5.4.10 Widgets

Widgets override the way a field is rendered in a create/update form. T2 defines two widgets:

```
T2.rating_widget(self,value,callback=None)
T2.tag_widget(self,value,tags=[])
```

The first of them is used by `t2.ratings`. The latter allows multiple values to be stored in a field. Here is an example:

```
db.mytable.myfield=T2.tag_widget(['red','green','blue'])
```

Values are stored as "[red][green]".

5.4.11 Google Checkout (experimental)

```
t2.clear_cart(self)
t2.add_to_cart(self,name,price,quantity=1,description='',weight
    =0,height=0,length=0,depth=0,currency='USD',weight_unit='LB')
t2.checkout_cart(self,merchant_id,action_url,button_url,
    continue_url,attributes={})
```

The first clears the cart (session.t2.cart). The second adds items to the cart. The latter generates a form with hidden buttons and a "Google check-out" button. On pressing the button, the control goes to Google Checkout and the visitor can pay using the credit card. For details look at the Google Checkout docs.

5.4.12 Other Issues

T2 uses and defines the following tables:

- `t2_person`
- `t2_group`
- `t2_membership`
- `t2_access`
- `t2_attachment`
- `t2_comment`
- `t2_review`

The user can override these tables by defining them before instantiating T2. They must have at least the fields defined in the default.

User defined tables used with T2 CRUD will always be STAMPED if the tables have the following fields

- `created_by_ip`
- `created_on`
- `created_by`
- `created_signature`
- `modified_by_ip`
- `modified_on`
- `modified_by`
- `modified_signature`

The `created_by` stores the id of the person creating the record. `created_signature` stores the name of the person who created the record and avoids unnecessary joins. `created_by_ip` is the IP address of the client who created the record. Yes this is redundant but it is a trick to avoid joins and make everything work on GAE.

If the field `modified_on` is present, it is used by t2.update() to detect conflict and notify the visitor when multiple clients are trying to update the same record.

T2 also relies on additional special table and field attributes to customize forms:

- `db.mytable.myfield.label` is the label to be used in forms.
- `db.mytable.myfield.comment` is the comment to be shown in the third column of create/update forms.
- `db.mytable.exposes` is a list of fields that should appear in create/update forms (None means all fields).
- `db.mytable.displays` is a list of fields that should appear in display forms (None, means all fields).
- `db.mytable.represent` is a lambda function that takes a row for the table and return a representation for that row.

6 Files in T2

```
__init__.py
ABOUT
cache
controllers
controllers/appadmin.py
controllers/default.py
databases
errors
languages
LICENSE
models
modules
modules/__init__.py
```

```
modules/t2.py
private
README
sessions
static
static/default
static/default/media
static/default/media/bl.gif
static/default/media/bl.png
static/default/media/bm.gif
static/default/media/bm.png
static/default/media/br.gif
static/default/media/br.png
static/default/media/closebox.gif
static/default/media/closebox.png
static/default/media/img01.jpg
static/default/media/img02.jpg
static/default/media/img03.jpg
static/default/media/img04.jpg
static/default/media/img05.jpg
static/default/media/metal.png
static/default/media/ml.gif
static/default/media/ml.png
static/default/media/mr.gif
static/default/media/mr.png
static/default/media/spacer.gif
static/default/media/t2.png
static/default/media/t2_192.png
static/default/media/title.png
static/default/media/tl.gif
static/default/media/tl.png
static/default/media/tm.gif
static/default/media/tm.png
static/default/media/tr.gif
static/default/media/tr.png
static/default/scripts
static/default/scripts/calendar.js
static/default/scripts/fancyzoom.min.js
static/default/scripts/jquery.js
static/default/scripts/sfmenu.js
static/default/styles
static/default/styles/calendar.css
static/default/styles/sfmenu.css
static/default/styles/style.css
tests
```

```
uploads
views
views/appadmin.html
views/default
views/default/create_puppy.html
views/default/display_puppy.html
views/default/login.html
views/default/register.html
views/generic.html
views/head.html
views/layout.html
```