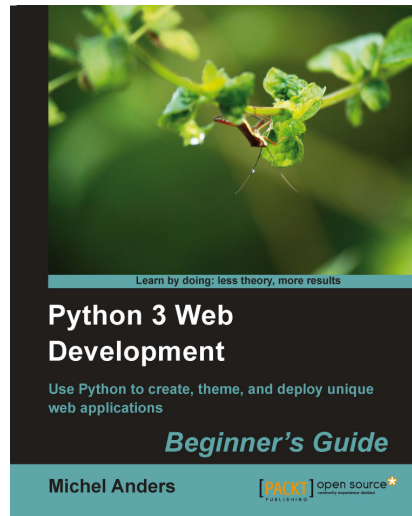


Python 3 Web Development Beginner's Guide

Michel Anders



Chapter No. 3 "Tasklist I: Persistence"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Tasklist I: Persistence"

A synopsis of the book's content

Information on where to buy this book

About the Author

Michel Anders, after his chemistry and physics studies where he spent more time on computer simulations than on real world experiments, the author found his real interests lay with IT and Internet technology, and worked as an IT manager for several different companies, including an Internet provider, a hospital, and a software development company.

After his initial exposure to Python as the built-in scripting language of Blender, the popular 3D modeling and rendering suite, the language became his tool of choice for many projects.

He lives happily in a small converted farm, with his partner, three cats, and twelve goats. This tranquil environment proved to be ideally suited to writing his first book, *Blender 2.49 Scripting* (Packt Publishing, 978-1-849510-40-0).

He loves to help people with Blender and Python-related questions and may be contacted as 'varkenvarken' at <http://www.blenderartists.org/> and maintains a blog on Python-specific subjects at <http://michelanders.blogspot.com/>.

For Clementine, always.

For More Information:

www.packtpub.com/python-3-web-development-beginners-guide/book

Python 3 Web Development Beginner's Guide

Building your own Python web applications provides you with the opportunity to have great functionality, with no restrictions. However, creating web applications with Python is not straightforward. Coupled with learning a new skill of developing web applications, you would normally have to learn how to work with a framework as well.

Python 3 Web Development Beginner's Guide shows you how to independently build your own web application that is easy to use, performs smoothly, and is themed to your taste—all without having to learn another web framework.

Web development can take time and is often fiddly to get right. This book will show you how to design and implement a complex program from start to finish. Each chapter looks at a different type of web application, meaning that you will learn about a wide variety of features and how to add them to your customized web application. You will also learn to implement jQuery into your web application to give it extra functionality. By using the right combination of a wide range of tools, you can have a fully functional, complex web application up and running in no time.

A practical guide to building and customizing your own Python web application, without the restriction of a pre-defined framework.

What This Book Covers

Chapter 1, Choosing Your Tools, looks at the many aspects of designing web applications. The idea is to provide you with an overview that may help you recognize components in subsequent chapters and give you some insight into the arguments used to decide which tool or library to use. We also illustrate some issues that are relevant when designing an application that does not deal with coding directly, such as security or usability.

Chapter 2, Creating a Simple Spreadsheet, develops a simple spreadsheet application. The spreadsheet functionality will be entirely implemented in JavaScript plus jQuery UI, but on the server-side, we will encounter the application server, CherryPy, for the first time and we will extend it with Python code to deliver the page that contains the spreadsheet application dynamically.

Chapter 3, Tasklist I: Persistence, a full fledged web application needs functionality to store information on the server and a way to identify different users. In this chapter, we address both issues as we develop a simple application to maintain lists of tasks.

For More Information:

www.packtpub.com/python-3-web-development-beginners-guide/book

Chapter 4, Tasklist II: Databases and AJAX, refactors the tasklist application developed in the previous chapter. We will use the SQLite database engine on the server to store items and will use jQuery's AJAX functionality to dynamically update the contents of the web application. On the presentation side, we will encounter jQuery UI's event system and will learn how to react on mouse clicks.

Chapter 5, Entities and Relations, most real life applications sport more than one entity and often many of these entities are related. Modeling these relations is one of the strong points of a relational database. In this chapter, we will develop a simple framework to manage these entities and use this framework to build an application to maintain lists of books for multiple users.

Chapter 6, Building a Wiki, develops a wiki application and in doing so we focus on two important concepts in building web applications. The first one is the design of the data layer. The wiki application is quite complex, and in this chapter, we try to see where the limitations in our simple framework lie. The second one is input validation. Any application that accepts input from all over the Internet should check the data it receives, and in this chapter, we look at both client-side and server-side input validation.

Chapter 7, Refactoring Code for Reuse, after doing a substantial bit of work, it is often a good idea to take a step back and look critically at your own work to see if things could have been done better. In this chapter, we look at ways to make the entity framework more generally useful and employ it to implement the books application a second time.

Chapter 8, Managing Customer Relations, there is more to an entity framework and CherryPy application code than merely browsing lists. The user must be able to add new instances and edit existing ones. This chapter is the start of the development of a CRM application that will be extended and refined in the final chapters.

Chapter 9, Creating Full-Fledged Webapps: Implementing Instances, focuses on the design and implementation of the user interface components to add and maintain entities, and relations between entities, in a way that is independent of the type of entity. This functionality is immediately put to use in the CRM application that we develop. Managing user privileges is another issue we encounter as we explore the concept of role-based access control.

Chapter 10, Customizing the CRM Application, is the final chapter and it extends our framework and thereby our CRM application by taking a look at browsing, filtering, and sorting large numbers of entities. We also take a look at what is needed to allow customization by the end user of the application's appearance and its functionality.

Appendix A, References to Resources, is a convenient overview of both Web and paper resources.

For More Information:

www.packtpub.com/python-3-web-development-beginners-guide/book

3

Tasklist I: Persistence

In the previous chapter, we learned how to deliver content to the user. This content consisted of HTML markup to structure the information together with a number of JavaScript libraries and code to create a user interface.

We noted that this was not a full-fledged web application yet; it lacked the functionality to store information on the server and there was no way to identify different users or any way to authenticate them. In this chapter, we will address both these issues when we design a simple tasklist application.

This tasklist application will be able to serve multiple users and store the list of tasks for each user on the server.

Specifically, we will look at:

- ◆ How to design a tasklist application
- ◆ How to implement a logon screen
- ◆ What a session is and how this allows us to work with different users at the same time
- ◆ How to interact with the server and add or delete tasks
- ◆ How to make entering dates attractive and simple with jQuery UI's datapicker widget
- ◆ How to style button elements and provide tooltips and inline labels to input elements

Designing a tasklist application

Designing an application should start with a clear idea of what is expected. Not only to determine what is technically required, but almost as important, to define clear boundaries so that we don't lose time on things that are just nice to have. Nice to have features are something to be added if there is time left in the project.

For More Information:

www.packtpub.com/python-3-web-development-beginners-guide/book

So let's draw up a shortlist of the relevant features of our tasklist application. Some of these may seem obvious, but as we will see, these have a direct impact on some implementation choices that we have to make, such as:

- ◆ The application will be used by multiple users
- ◆ Task lists should be stored indefinitely
- ◆ A task list may contain an unlimited number of tasks but the user interface is designed for optimal performance for up to 25 tasks or so
- ◆ Tasks may be added, deleted, and marked as done

Although this list isn't exhaustive, it has some important implications.

The fact that the tasklist application will be used by more than one user means that we have to identify and authorize people who want to use it. In other words, we will need some sort of logon screen and a way to check people against some sort of password database. Because we do not want to burden the user with identifying himself/herself each and every time a task list is refreshed or altered, we need some way of implementing the concept of a **session**.

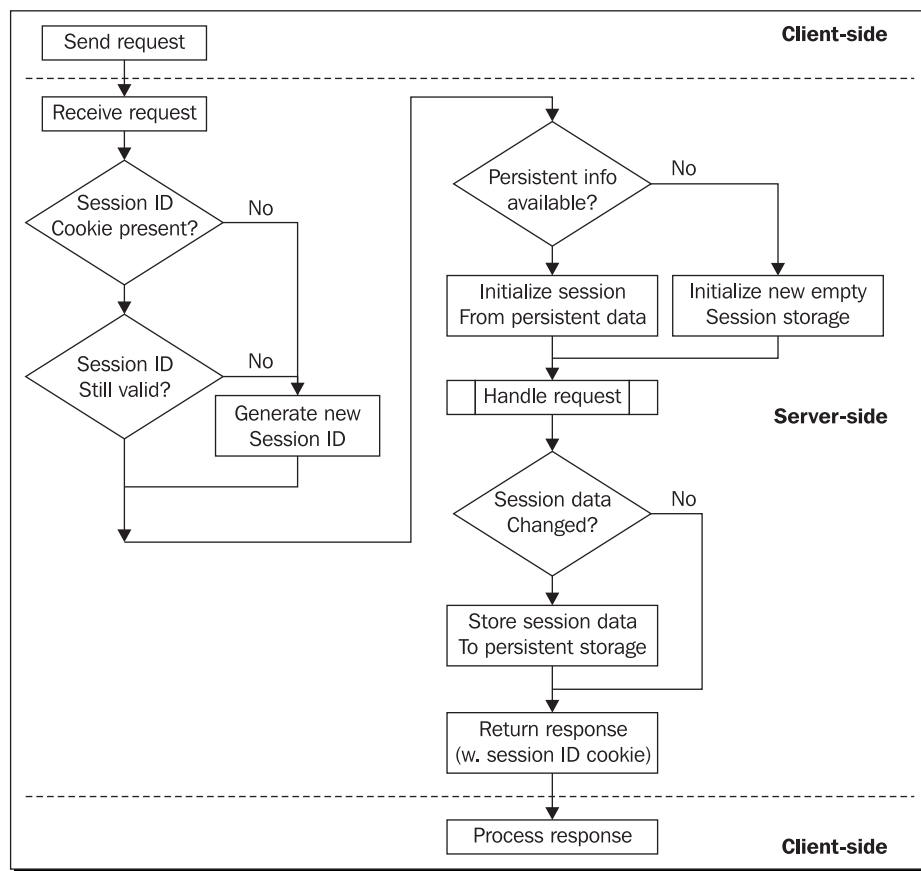
Web applications use the stateless HTTP protocol. This means, from the server's point of view, every request is a single, unrelated event, and no information is retained at the server. This obviously presents us with a problem if we want to perform a set of related actions. The solution is to ask the web browser to send a small piece of information along with every request it makes to the application after the application has identified the user.

This might be accomplished in a number of ways. The server may add an extra parameter to all links inside any web page it generates, commonly referred to as a **session id**, or use the even more general concept of a **cookie**.

Once the server asks the web browser to store a cookie, this cookie is sent with every following request to the same website. The advantage of cookies is that common web application frameworks (like CherryPy) are already equipped to deal with them and implementing sessions with cookies is much simpler than designing the application to alter all hyperlinks it generates to include a proper session ID. The disadvantage might be that people may block their browser from storing cookies because some websites use them to track their clicking behavior.

We let the simplicity of implementation prevail and opt for cookies. If users want to block cookies this is not much of a problem as most browsers also have the option to selectively allow cookies from designated websites.

The following image illustrates the way CherryPy manages sessions with the help of cookies:



It starts when the client (the web browser) sends a request to CherryPy. Upon receiving the request, the first check is to see if the web browser has sent along a cookie with a session ID. If it didn't, a new session ID is generated. Also, if there was a cookie with a session ID, if this ID is no longer valid (because it has expired, for example, or is a remnant from a very old interaction and doesn't exist in the current cache of session IDs) CherryPy also generates a new session ID.

At this point, no persistent information is stored if this is a new session, but if it's an existing session there might be persistent data available. If there is, CherryPy creates a `Session` object and initializes it with the available persistent data. If not, it creates an empty `Session` object. This object is available as a global variable `cherrypy.session`.

The next step for CherryPy is to pass control to the function that will handle the request. This handler has access to the `Session` object and may change it, for example, by storing additional information for later reuse. (Note that the `Session` object acts like a dictionary so you can simply associate values with keys with `cherrypy.session['key']=value`. The only restriction to the keys and values is that they must be serializable if the persistent storage is on disk).

Then before returning the results generated by the handler, CherryPy checks if the `Session` object has changed. If (and only if) it has, are the contents of the `Session` object saved to a more permanent storage.

Finally, the response is returned accompanied by a cookie with the session ID.

Time for action – creating a logon screen

Our first task is to create a small application that does little more than present the user with a logon screen. It will be the starting point of our tasklist application and many others as well.

The code for this example as well as most other examples in this book is available from the Packt website. If you have not downloaded it yet, this might be a good time to do so.

Enter the following pieces of code and save it in a file called `logonapp.py` in the same directory as the other files distributed with this chapter (*Chapter 3* in the sample code):

Chapter3/logonapp.py

```
import cherrypy
import logon

class Root(object):
    logon = logon.Logon(path="/logon",
                        authenticated="/",
                        not_authenticated="/goaway")

    @cherrypy.expose
    def index(self):
        username=logon.checkauth('/logon')
        return '''
        <html><body>
        <p>Hello user <b>%s</b></p>
        </body></html>'''%username

    @cherrypy.expose
    def goaway(self):
        return '''
        <html>
```

```

        <body><h1>Not authenticated, please go away.</h1>
    </body></html>'''

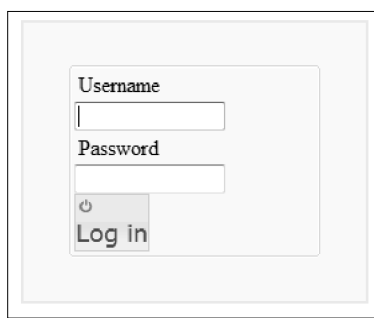
    @cherry.py.expose
    def somepage(self):
        username=logon.checkauth('/logon',returntopage=True)
        return '''<html>
            <body><h1>This is some page.</h1>
            </body>
            </html>'''

if __name__ == "__main__":
    import os.path
    current_dir = os.path.dirname(os.path.abspath(__file__))

    cherry.py.quickstart(Root(),config={
        '/': {'tools.sessions.on': True }
    })

```

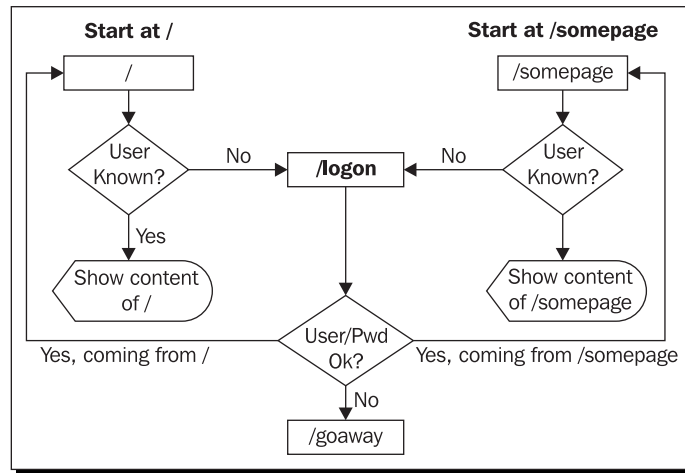
If you now run `logonapp.py`, a very simple application is available on port 8080. It presents the user with a logon screen when the top level page `http://localhost:8080/` is accessed. An example is shown in the following illustration:



If a correct username/password combination is entered, a welcome message is shown. If an unknown username or wrong password is entered, the user is redirected to `http://localhost:8080/goaway`.

The `somepage()` method (highlighted) returns a page with (presumably) some useful content. If the user is not yet authenticated, the logon screen is shown and upon entering the correct credentials, the user is directed back to `http://localhost:8080/somepage`.

The complete tree of web pages within the logon sample application and the possible paths the user may pick through is shown next:



Logon + session ID vs. HTTP basic authentication

You may wonder why we choose not to reuse CherryPy's bundled `auth_basic` tool that offers basic authentication (for more information on this tool, see http://www.cherrypy.org/wiki/BuiltinTools#tools.auth_basic). If all we wanted was to check whether a user is allowed access to a single page, this would be a good choice. The basic authentication is sufficient to authenticate a user, but has no concept of a session. This means we lack a way to store data that needs to be accessible when we process subsequent requests by the same user. The `sessions` tool we use here does provide this additional functionality.

What just happened?

Part of the magic of `logonapp.py` is achieved by enabling the 'sessions' tool in CherryPy. This is what is done by passing the `tools.sessions.on` key with `True` as a value to the configuration dictionary for the `quickstart()` function.

However, most of the hard work in `logonapp.py` is actually performed by the module `logon`:

Chapter3/logon.py

```
import cherrypy
import urllib.parse

def checkauth(logonurl="/", returntopage=False):
    returnpage=''
```

```

        if returntopage:
            returnpage='?returnpage='
                + cherry.py.request.script_name
                + cherry.py.request.path_info

        auth = cherry.py.session.get('authenticated',None)
        if auth == None :
            raise cherry.py.HTTPRedirect(logonurl+returnpage)
        return auth

class Logon:
    base_page = '''
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<script type="text/javascript" src="/jquery.js" ></script>
<script type="text/javascript" src="/jquery-ui.js" ></script>
<style type="text/css" title="currentStyle">
    @import "/jquerytheme.css";
    @import "/static/css/logon.css";
</style>
</head>
<body id="logonscreen">
<div id="content">
    %s
</div>
<script type="text/javascript">$("#button").button({icons: {primary:
'ui-icon-power'}})</script>
</body>
</html>
'''

    logon_screen = base_page % '''
<form class="login" action="%s/logon" method="GET">
<fieldset>
<label for="username">Username</label>
<input id="username" type="text" name="username" />
<script type="text/javascript">$("#username").focus()</script>
<label for="password">Password</label>
<input id="password" type="password" name="password" />
<input type="hidden" name="returnpage" value="%s" />
<button type="submit" class="login-button" value="Log in">
Log in
</button>
</fieldset>

```

```
</form>
'''

not_authenticated =
    base_page % '''<h1>Login or password not correct</h1>'''

def __init__(self, path="/logon",
              authenticated="/", not_authenticated="/"):
    self.path=path
    self.authenticated=authenticated
    self.not_authenticated=not_authenticated

    @staticmethod
    def checkpass(username,password):
        if username=='user' and password=='secret': return True
        return False

    @cherrypy.expose
    def index(self,returnpage=''):
        return Logon.logon_screen % (
            self.path,urllib.parse.quote(returnpage))

    @cherrypy.expose
    def logon(self,username,password,returnpage=''):
        returnpage = urllib.parse.unquote(returnpage)
        if Logon.checkpass(username,password):
            cherrypy.session['authenticated']=username
            if returnpage != '':
                raise cherrypy.InternalRedirect(returnpage)
            else:
                raise cherrypy.InternalRedirect(
                    self.authenticated)
        raise cherrypy.InternalRedirect(self.not_authenticated)

    @cherrypy.expose
    def logoff(self,logoff):
        cherrypy.lib.sessions.expire()
        cherrypy.session['authenticated']=None
        raise cherrypy.InternalRedirect(self.not_authenticated)
```

The logon module implements a utility function `checkauth()` (highlighted). This function is designed to be called from anywhere inside a CherryPy application. If the user is already authenticated, it will return the username; otherwise it will redirect the user to a URL that should present the user with a logon screen. If the `returnpage` parameter is true, this URL is augmented with an extra parameter `returnpage` containing the URL of the page that invoked `checkauth()`. The logon page (or rather the handler implementing it) should be designed to redirect the user to the URL in this parameter if the authentication is successful.

As we have seen, typical use for the `checkauth()` function would be to call it from every page handler that serves content that requires authentication.

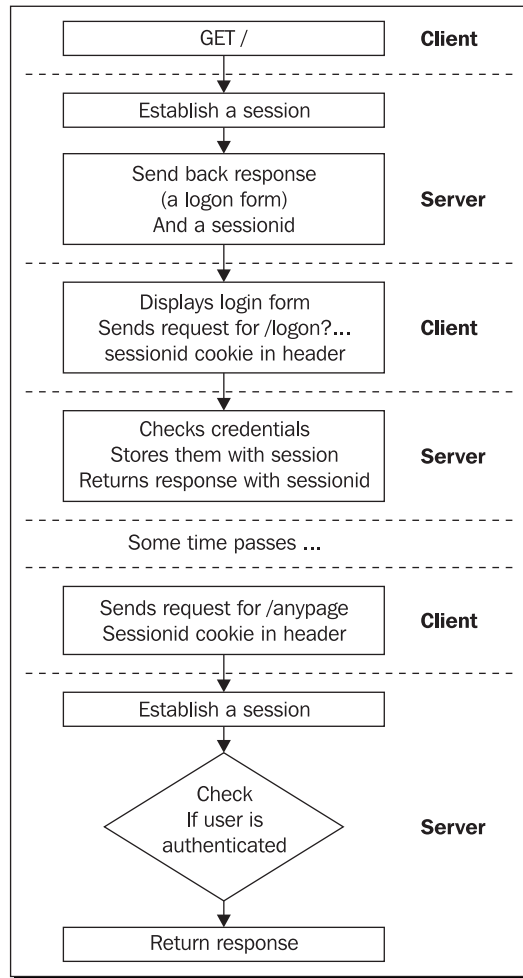
`checkauth()` itself does just two things: First it determines the page to return to (if necessary) by concatenating the `script_name` and `path_info` attributes from the `cherry.py.request` object that CherryPy makes available. The first one contains the path where a CherryPy tree is mounted, the last one contains the path within that tree. Together they form the complete path to the handler that invoked this `checkauth()` function.

The second thing that `checkauth()` does is it determines whether `cherry.py.session` (a dictionary like Session object) contains an `authenticated` key. If it does, it returns the associated value, if not, it redirects to the logon page.

The `cherry.py.session` variable is a `cherry.py.lib.sessions.Session` object available to each request. It acts like a dictionary and initially it is devoid of any keys. When a value is assigned to the first new key, a persistent object is created that is associated with the session ID and upon finishing a request, the `Session` object is stored and its session ID is passed as the value of a `session_id` cookie in the response headers. If a subsequent request contains a request header with a `session_id` cookie, a `Session` object with the corresponding session ID is retrieved from storage, making any saved key/value pairs available again.

The default storage scheme is to keep data in memory. This is fast and simple but has the disadvantage that restarting the CherryPy server will discard this data, effectively expiring all sessions. This might be ok for short-lived sessions, but if you need a more persistent solution, it is possible to store the session information as files (by setting the `tools.sessions.storage_type` configuration key to "file") or even to a database backend. For more about sessions, see CherryPy's online documentation on the subject at <http://cherry.py.org/wiki/CherryPySessions>.

The various steps in the communication between the client and the server during a session are shown in the following illustration:



The bulk of the `logon` module is provided by the `Logon` class. It implements several methods (these methods are highlighted in the code listed on the previous pages as well):

- ◆ `__init__()` will initialize a `Logon` instance to hold the path to the point where this `Logon` instance is mounted on the tree of handlers, together with the default URLs to redirect to successful and unsuccessful authentication.
- ◆ `checkpass()` is a static function that takes a username and a password and returns `True` if these are a matching pair. It is designed to be overridden by a more suitable definition.

Logon also exposes three handler methods to the CherryPy engine:

- ◆ `index()` is a method that will serve the actual logon screen
- ◆ `logon()` is passed the username and password when the user clicks on the logon button
- ◆ `logout()` will expire a session, causing subsequent calls to `checkauth()` to redirect the user to the logon screen

The `Logon` class also contains a number of class variables to hold the HTML presented by the `index()` method. Let's look at the methods in detail.



And what about security? The `Logon` class we design here has no facilities to prevent people from eavesdropping if they have access to the wire that transports the HTTP traffic. This is because we transmit the passwords unencrypted. We may implement some sort of encryption scheme ourselves, but if your design requires some form of protection, it is probably better and easier to communicate over a secure HTTPS channel. CherryPy may be configured to use HTTPS instead of HTTP. More on it can be found at: <http://cherrypy.org/wiki/ServerObject>.

Pop quiz – session IDs

1. If the client sends a new session ID again and again, wouldn't that fill up all storage on the server eventually?
2. If the client has cookies disabled, what happens to the generation of session IDs?
 - a. The server will stop generating new session IDs, returning the same ID repeatedly
 - b. The server will stop returning new session IDs
 - c. The server will keep generating and returning new session IDs

Serving a logon screen

The `index()` method serves the HTML to present the user with a logon screen. At its core, this HTML is a `<form>` element with three `<input>` elements: a regular text input where the user may enter his/her username, a password input (that will hide the characters that are entered in this field), and an `<input>` element that has a `hidden` attribute. The `<form>` element has an `action` attribute that holds the URL of the script that will process the variables in the form when the user clicks the logon button. This URL is constructed to point to the `logon()` method of our `Logon` class by appending `/logon` to the path that the `Logon` instance was mounted on in the CherryPy tree.

The `<input>` element we marked as hidden is initialized to hold the URL that the user will be redirected to when `logon()` authenticates the user successfully.

The form that makes up the logon screen also contains a tiny piece of JavaScript:

```
$("#username").focus()
```

It uses jQuery to select the input element that will receive the username and gives it focus. By placing the cursor in this field, we save the user the effort of pointing and clicking on the username field first before the username can be entered. Now he can start typing right away. Note that this code snippet is not placed near the end of the document, but right after the `<input>` element to ensure execution as soon as the `<input>` element is defined. The logon page is so small that this might be irrelevant, but on slow loading pages, key presses might be misdirected if we waited to shift the focus until the whole page had loaded.



Be aware that the logon form we construct here has a `<form>` element with an `action="GET"` attribute. This works fine, but has a disadvantage: parameters passed with a GET method are appended to the URL and may end up in the log files of the server. This is convenient when debugging, but you might not want that for a production environment, as this might leave passwords exposed. The `action` attribute can be changed to `POST` though without any change to the Python code handling the request as CherryPy takes care of the details. Parameters passed to a POST method are not logged, so a POST method might be better suited to a password verification request.

Setting up a session

The `logon()` method is passed the contents of all the `<input>` elements in the form as parameters. The `username` and `password` parameters are passed to the `checkpass()` method and if the user's credentials are right, we establish a session by associating the username with the authenticated key in our session storage with `cherrypy.session['authenticated']=username`.

This will have the effect that every response sent to the browser will contain a cookie with a session ID and any subsequent request to CherryPy that contains this cookie again will cause the handler for that request to have access to this same session storage.

After successful authentication, `logon()` redirects the user to the return page if one was passed to it or to the default page passed to it upon initialization of the `Logon` instance. If authentication fails, the user is redirected to a non-authorized page.

Expiring a session

The `logoff()` method is provided to offer a possibility to actively expire a session. By default, a session expires after 60 minutes, but the user might want to sign off explicitly, either to make sure that no one sneaks behind his keyboard and continues in his name or to log on as a different persona. Therefore, you will find, in most applications, a discrete logoff button, often positioned in the upper-right corner. This button (or just a link) must point to the URL that is handled by the `logoff()` method and will cause the session to be invalidated immediately by removing all session data.

Note that we have to take special precautions to prevent the browser from caching the response from the `logoff()` method, otherwise it may simply redisplay the response from the last time the logoff button was pressed without actually causing `logoff()` to be called. Because `logoff()` always raises an `InternalRedirect` exception, the actual response comes from a different source. This source, for example, the `goaway()` method in the `Root` class must be configured to return the correct response headers in order to prevent the web browser from caching the result. This is accomplished by configuring the `goaway()` method in `logonapp.py` with CherryPy's expires tool like the following:

Chapter3/logonapp.py

```
@cherry.py.expose
def goaway(self):
    return '''
<html><body>
<h1>Not authenticated, please go away.</h1>
</body></html>
'''
    goaway._cp_config = {
        'tools.expires.on': True,
        'tools.expires.secs': 0,
        'tools.expires.force': True}
```

The highlighted line is where we configure the handler (the `goaway()` method) to set expiration headers in the response by assigning a configuration dictionary to the `_cp_config` variable.



Assigning to a variable that is part of a function might seem odd, but functions and methods in Python are just objects and any object may have variables. New variables might be assigned to an object even after its definition. Upon calling a handler, CherryPy checks if that handler has a `_cp_config` variable and acts accordingly. Note that the `@cherry.py.expose` decorator also merely sets the `expose` variable on the handler to `true`.

Have a go hero – adding a logon screen to the spreadsheet application

In the previous chapter, we had created an application that serves a spreadsheet. If you wanted to serve this spreadsheet only to authenticated users, what would we have to change to use the logon module presented in the previous section?

Hint: You need to do three things, one involves mounting an instance of the `Logon` class on the CherryPy tree, the other is changing the handler that serves the spreadsheet to check for authentication, and finally you need to enable sessions.

An example implementation is available as `spreadsheet3.py`.

Designing a task list

Now that we have looked at ways to authenticate the users, let's look at the implementation of the task list itself.

A task list would be unusable if its contents evaporated once the browser was closed. We therefore need some way to persistently store these task lists. We could use a database and many of the example applications in this book do use a database to store data. For this application, we will opt to use the filesystem as a storage medium, simply storing tasks as files containing information about a task, with separate directories for each user. If we dealt with huge amounts of users or very long task lists, the performance of such an implementation probably wouldn't suffice, but by using simple files for storage, we won't have to design a database schema which saves us quite some time.

By limiting ourselves to fairly short task lists, our user interface may be kept relatively simple as there will be no need for pagination or searching. This doesn't mean the user interface shouldn't be easy to use! We will incorporate jQuery UI's **datepicker** widget to assist the user with choosing dates and will add tooltips to user interface components to provide a shallow learning curve of our task list application.

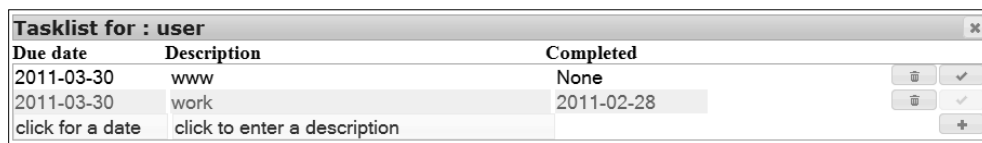
The final requirements more or less define what we understand a task to be and what we are supposed to do with it: A task has a description and a due date and because it can be marked as done, it should be able to store that fact as well. Furthermore, we limit this application to adding and deleting tasks. We explicitly do not provide any way to alter a task, except for marking it as done.

Time for action – running tasklist.py

Let's first have a look at what the application looks like:

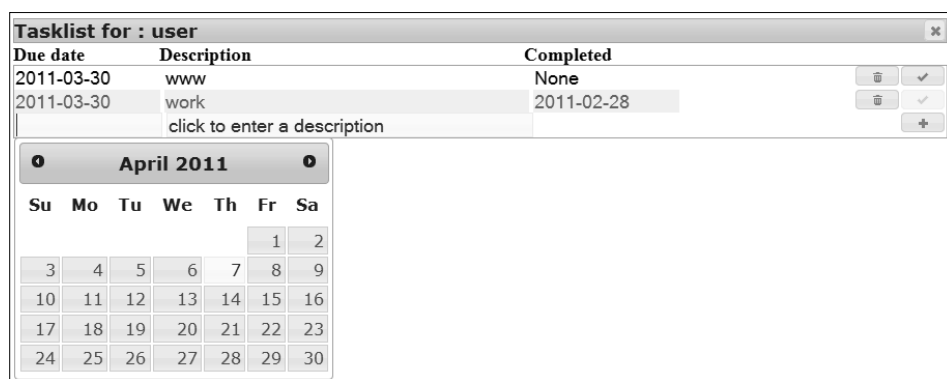
1. Start up `tasklist.py` from the code directory of this chapter.
2. Point your browser to `http://localhost:8080`.

3. In the login screen, enter **user** as the username and **secret** as the password.
4. You are now presented with a rather stark looking and empty task list:



Due date	Description	Completed		
2011-03-30	www	None		
2011-03-30	work	2011-02-28		
click for a date	click to enter a description			

You should be able to add a new task by entering a date and a description in the input boxes and pressing the add button. Entering a date is facilitated by jQuery UI's datepicker widget that will pop up once you click the input field for the date, as shown in the following screenshot:



Due date	Description	Completed		
2011-03-30	www	None		
2011-03-30	work	2011-02-28		
	click to enter a description			

April 2011

Su	Mo	Tu	We	Th	Fr	Sa
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30

Once you have added one or more tasks, you can now either delete those tasks by clicking the button with the little trash can icon or mark it as done by clicking the button with the check icon. Tasks marked as done have a slightly different background color depending on the chosen theme. If you mark a task as done, its completion date will be today. You can select a different date by clicking on the completion date of a task (displayed as **None** for an unfinished task). It will present you with yet another datepicker, after which the selected date will be stored as the completion date once the done button is clicked. The following screenshot gives an impression of a task list with numerous items:



Due date	Description	Completed		
2010-08-28	work	2010-08-28		
2010-08-29	more work	2010-08-29		
2010-08-30	even more work	None		
2010-08-31	relax	None		
2010-09-08	work again	None		
2010-09-09	and again	None		
2010-09-10	and again			

click to enter a description

There is some hidden magic that might not be immediately obvious. First of all, all the tasks are sorted according to their **Due date**. This is done on the client-side with the help of some JavaScript and a jQuery plugin, as we will see in the section on JavaScript. Also accomplished with some JavaScript are the tooltips. Both hovering tooltips on every button and the inline help text inside the `<input>` elements are added with the same script. We will examine this in depth.

What just happened?

`tasklist.py` is rather straightforward as it delegates most work to two modules: the `logon` module that we encountered in the previous sections and a `task` module that deals with displaying and manipulating task lists.

The highlighted line in the following code shows the core of the application. It starts up CherryPy with a suitable configuration. Note that we enabled the sessions tool, so that we can actually use the `logon` module. Also, we construct the path to jQuery UI's theme stylesheet in such a way that it depends on the `theme` variable to make changing the application's theme simple (second highlight).

The instance of the `Root` class that we pass to `quickstart()` creates a simple tree:

```
/
/logon
/logon/logon
/logon/logoff
/task
/task/add
/task/mark
```

The top level URL `/` returns the same content as `/login` by calling the `index()` method of the `Logon` instance. We could have used an `InternalRedirect` exception, but this is just as simple. The paths starting with `/task` are all handled by an instance of the `Task` class:

Chapter3/tasklist.py

```
import cherrypy
import os.path
import logon
import task

current_dir = os.path.dirname(os.path.abspath(__file__))
theme = "smoothness"

class Root(object):
    task = task.Task(logoffpath="/logon/logoff")
    logon = logon.Logon(path="/logon",
```

```

        authenticated="/task",
        not_authenticated="/")

    @cherrypy.expose
    def index(self):
        return Root.logon.index()

if __name__ == "__main__":
    cherrypy.quickstart(Root(), config={
        '/':
        { 'log.access_file': os.path.join(current_dir, "access.log"),
          'log.screen': False,
          'tools.sessions.on': True
        },
        '/static':
        { 'tools.staticdir.on': True,
          'tools.staticdir.dir': os.path.join(current_dir, "static")
        },
        '/jquery.js':
        { 'tools.staticfile.on': True,
          'tools.staticfile.filename': os.path.join(current_dir,
            "static", "jquery", "jquery-1.4.2.js")
        },
        '/jquery-ui.js':
        { 'tools.staticfile.on': True,
          'tools.staticfile.filename': os.path.join(current_dir,
            "static", "jquery", "jquery-ui-1.8.1.custom.min.js")
        },
        '/jquerytheme.css':
        { 'tools.staticfile.on': True,
          'tools.staticfile.filename': os.path.join(current_dir,
            "static", "jquery", "css", theme, "jquery-ui-1.8.4.custom.css")
        },
        '/images':
        { 'tools.staticdir.on': True,
          'tools.staticdir.dir': os.path.join(current_dir,
            "static", "jquery", "css", theme, "images")
        }
    })

```

Python: the task module

The `task` module is implemented in the file `task.py`. Let's look at the parts that make up this file.

Time for action – implementing the task module

Have a look at the Python code in `task.py`:

Chapter3/task.py

```
import cherryypy
import json

import os
import os.path
import glob
from configparser import RawConfigParser as configparser
from uuid import uuid4 as uuid
from datetime import date

import logon
```

This first part illustrates Python's "batteries included" philosophy nicely: besides the `cherryypy` module and our own `logon` module, we need quite a bit of specific functionality. For example, to generate unique identifiers, we use the `uuid` module and to manipulate dates, we use the `datetime` module. All of this functionality is already bundled with Python, saving us an enormous amount of development time. The next part is the definition of the basic HTML structure that will hold our task list:

Chapter3/task.py

```
base_page = '''
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/
TR/html4/strict.dtd">
<html>
<head>
<script type="text/javascript" src="/jquery.js" ></script>
<script type="text/javascript" src="/jquery-ui.js" ></script>
<style type="text/css" title="currentStyle">
    @import "/static/css/tasklist.css";
    @import "/jquerytheme.css";
</style>
<script type="text/javascript" src="/static/js/sort.js" ></script>
<script type="text/javascript" src="/static/js/tooltip.js" ></script>
<script type="text/javascript" src="/static/js/tasklist.js" ></script>
</head>
<body id="%s">
<div id="content">
%s
</div>
</body>
```

```
</html>
'''
```

Again the structure is simple, but besides the themed stylesheet needed by jQuery UI (and reused by the elements we add to the page), we need an additional stylesheet specific to our task list application. It defines specific layout properties for the elements that make up our task list (first highlight). The highlighted `<script>` elements show that besides the jQuery and jQuery UI libraries, we need some additional libraries. Each of them deserves some explanation.

What just happened?

The first JavaScript library is `sort.js`, a code snippet from James Padolsey (<http://james.padolsey.com/tag/plugins/>) that provides us with a plugin that allows us to sort HTML elements. We need this to present the list of tasks sorted by their due date.

The second is `tooltip.js` that combines a number of techniques from various sources to implement tooltips for our buttons and inline labels for our `<input>` elements. There are a number of tooltip plugins available for jQuery, but writing our own provides us with some valuable insights so we will examine this file in depth in a later section.

The last one is `tasklist.js`. It employs all the JavaScript libraries and plugins to actually style and sort the elements in the task list.

The next part of `task.py` determines the directory we're running the application from. We will need this bit of information because we store individual tasks as files located relative to this directory. The `gettaskdir()` function takes care of determining the exact path for a given username (highlighted). It also creates the `taskdir` directory and a sub directory with a name equal to username, if these do not yet exist with the `os.makedirs()` function (notice the final 's' in the function name: this one will create all intermediate directories as well if they do not yet exist):

Chapter3/task.py

```
current_dir = os.path.dirname(os.path.abspath(__file__))

def gettaskdir(username):
    taskdir = os.path.join(current_dir, 'taskdir', username)
    # fails if name exists but is a file instead of a directory
    if not os.path.exists(taskdir):
        os.makedirs(taskdir)
    return taskdir
```

The `Task` class is where the handlers are defined that CherryPy may use to show and manipulate the task list. The `__init__()` method stores a path to a location that provides the user with a possibility to end a session. This path is used by other methods to create a suitable logoff button.

The `index()` method will present the user with an overview of all his/her tasks plus an extra line where a new task can be defined. As we have seen, each task is adorned with buttons to delete a task or mark it as done. The first thing we do is check whether the user is authenticated by calling the `checkauth()` function from our `logon` module (highlighted). If this call returns, we have a valid username, and with that username, we figure out where to store the tasks for this user.

Once we know this directory, we use the `glob()` function from the Python `glob` module to retrieve a list of files with a `.task` extension. We store that list in the `tasklist` variable:

Chapter3/task.py

```
class Task(object):
    def __init__(self, logoffpath="/logoff"):
        self.logoffpath=logoffpath

    @cherrypy.expose
    def index(self):
        username = logon.checkauth()
        taskdir = gettaskdir(username)
        tasklist = glob.glob(os.path.join(taskdir, '*.task'))
```

Next, we create a `tasks` variable that will hold a list of strings that we will construct when we iterate over the list of tasks. It is initialized with some elements that together form the header of our task list. It contains, for example, a small form with a logoff button and the headers for the columns above the list of tasks. The next step is to iterate over all files that represent a task (highlighted) and create a form with suitable content together with delete and done buttons.

Each `.task` file is structured in a way that is consistent with Microsoft Windows `.ini` files. Such files can be manipulated with Python's `configparser` module. The `.task` file is structured as a single `[task]` section with three possible keys. This is an example of the format:

```
[task]
description = something
duedate = 2010-08-26
completed = 2010-08-25
```

When we initialize a `configparser` object, we pass it a dictionary with default values in case any of these keys is missing. The `configparser` will read a file when we pass an open file descriptor to its `readfp()` method. The value associated with any key in a given section may then be retrieved with the `get()` method that will take a section and a key as parameters. If the key is missing, it supplies the default if that was provided upon initialization. The second highlighted line shows how this is used to retrieve the values for the `description` key.

Next, we construct a form for each `.task` file. It contains read-only `<input>` elements to display the **Due date**, **Description**, and the completion date plus buttons to delete the task or mark it as done. When these buttons are clicked the contents of the form are passed to the `/task/mark` URL (handled by the `mark()` method). The method needs to know which file to update. Therefore, it is passed a hidden value: the basename of the file. That is, the filename without any leading directories and stripped of its `.task` extension:

Chapter3/task.py

```

        tasks = [
'''
<div class="header">
Tasklist for user <span class="highlight">%s</span>
  <form class="logoff" action="%s" method="GET">
    <button type="submit" name="logoffurl"
      class="logoff-button" value="/">Log off
    </button>
  </form>
</div>
'''%(username,self.logoffpath),
'''
<div class="taskheader">
  <div class="left">Due date</div>
  <div class="middle">Description</div>
  <div class="right">Completed</div>
</div>
'''<div id="items" class="ui-widget-content">']

    for filename in tasklist:
        d = configparser(
            defaults={'description':'',
                    'duedate':'',
                    'completed':None})
        id = os.path.splitext(os.path.basename(filename))[0]
        d.readfp(open(filename))
        description = d.get('task','description')
        duedate = d.get('task','duedate')
        completed = d.get('task','completed')
        tasks.append(
'''
<form class="%s" action="mark" method="GET">
  <input type="text" class="duedate left"
    name="duedate" value="%s" readonly="readonly" />
  <input type="text" class="description middle"
    name="description" value="%s" readonly="readonly" />
'''

```

```
<input type="text" class="completed right editable-date tooltip"
      title="click to select a date, then click done"
      name="completed" value="%s" />
<input type="hidden" name="id" value="%s" />
<button type="submit" class="done-button"
        name="done" value="Done" >Done
</button>
<button type="submit" class="del-button"
        name="delete" value="Del" >Del
</button>
</form>
'''%('notdone' if completed==None else 'done',
    duedate,description,completed,id))
    tasks.append(
'''
<form class="add" action="add" method="GET">
  <input type="text" class="duedate left editable-date tooltip"
        name="duedate" title="click to select a date" />
  <input type="text" class="description middle tooltip"
        title="click to enter a description" name="description"/>
  <button type="submit" class="add-button"
        name="add" value="Add" >Add
  </button>
</form>
</div>
''')

    return base_page%('itemlist',"".join(tasks))
```

Finally, we append one extra form with the same type of input fields for **Due date** and **Description** but this time, not marked as read-only. This form has a single button that will submit the contents to the `/task/add` URL. These will be handled by the `add()` method. The actual content returned by the `index()` method consists of all these generated lines joined together and embedded in the HTML of the `base_page` variable.

Adding new tasks

New tasks are created by the `add()` method. Besides the value of the add button (which is not relevant), it will take a `description` and a `duedate` as parameters. To prevent accidents, it first checks if the user is authenticated, and if so, it determines what the `taskdir` for this user is.

We are adding a new task so we want to create a new file in this directory. To guarantee that it has a unique name, we construct this filename from the path to this directory and a globally unique ID object provided by Python's `uuid()` function from the `uuid` module. The `.hex()` method of a `uuid` object returns the ID as a long string of hexadecimal numbers that we may use as a valid filename. To make the file recognizable to us as a task file, we append the `.task` extension (highlighted).

Because we want our file to be readable by a `configparser` object, we will create it with a `configparser` object to which we add a `task` section with the `add_section()` method and `description` and `duedate` keys with the `set()` method. Then we open a file for writing and use the open file handle to this file within a context manager (the `with` clause), thereby ensuring that if anything goes wrong when accessing this file, it will be closed and we will proceed to redirect the user to that list of tasks again. Note that we use a relative URL consisting of a single dot to get us the index page. Because the `add()` method handles a URL like `/task/add` redirecting to `'.'` (the single dot), will mean the user is redirected to `/task/`, which is handled by the `index()` method:

Chapter3/task.py

```
@cherry.py.expose
def add(self, add, description, duedate):
    username = logon.checkauth()
    taskdir = gettaskdir(username)
    filename = os.path.join(taskdir, uuid().hex+'.task')
    d=configparser()
    d.add_section('task')
    d.set('task', 'description', description)
    d.set('task', 'duedate', duedate)
    with open(filename, "w") as file:
        d.write(file)
    raise cherry.py.InternalRedirect(" ")
```

Deleting a task

Deleting or marking a task as done are both handled by the `mark()` method. Besides an ID (the basename of an existing `.task` file), it takes `duedate`, `description`, and `completed` parameters. It also takes optional `done` and `delete` parameters, which are set depending on whether the done or delete buttons are clicked respectively.

Again, the first actions are to establish whether the user is authenticated and what the corresponding task directory is. With this information, we can construct the filename we will act on. We take care to check the validity of the `id` argument. We expect it to be a string of hexadecimal characters only and one way to verify this is to convert it using the `int()` function with 16 as the base argument. This way, we prevent malicious users from passing a file path to another user's directory. Even though it is unlikely that a 32 character random string can be guessed, it never hurts to be careful.

The next step is to see if we are acting on a click on the done button (highlighted in the following code). If we are, we read the file with a `configparser` object and update its `completed` key.

The `completed` key is either the date that we were passed as the `completed` parameter or the current date if that parameter was either empty or `None`. Once we have updated the `configparser` object, we write it back again to the file with the `write()` method.

Another possibility is that we are acting on a click on the delete button; in that case, the `delete` parameter is set. If so, we simply delete the file with the `unlink()` function from Python's `os` module:

Chapter3/task.py

```
@cherry.py.expose
def mark(self, id, due_date, description,
        completed, done=None, delete=None):
    username = logon.checkauth()
    taskdir = gettaskdir(username)
    try:
        int(id, 16)
    except ValueError:
        raise cherry.py.InternalRedirect(self.logoffpath)
    filename = os.path.join(taskdir, id + '.task')
    if done == "Done":
        d = configparser()
        with open(filename, "r") as file:
            d.readfp(file)
        if completed == "" or completed == "None":
            completed = date.today().isoformat()
        d.set('task', 'completed', completed)
        with open(filename, "w") as file:
            d.write(file)
    elif delete == "Del":
        os.unlink(filename)
    raise cherry.py.InternalRedirect(".")
```

JavaScript: tasklist.js

The buttons we present the end user need to be configured to respond to clicks in an appropriate manner and it would be nice if these buttons showed some intuitive icons as well. This is what we will take care of in `tasklist.js`.

Time for action – styling the buttons

The work done by `tasklist.js` is mainly concerned with styling the `<button>` elements and adding tooltips and inline labels to `<input>` elements. The results so far are shown in the following screenshot:

Due date	Description	Completed
2010-08-28	work	2010-08-28
2010-08-29	more work	2010-08-29
2010-08-30	even more work	None
2010-08-31	relax	None

click for a date click to enter a description +

What just happened?

As can be seen in the first line of `tasklist.js` (code starts on the next page), the work to be done is scheduled after loading the complete document by passing it to jQuery's `$(document).ready()` function.

The first step is to add to any element with a header class the `ui-widget` and `ui-widget-header` classes as well. This will cause these elements to be styled in a way that is consistent with the chosen theme.

Then we configure the add button (or rather any element with the `add-button` class) as a jQuery UI button widget. The option object passed to it will configure it to show no text, but just a single icon depicting a thick plus sign. We also add an extra function to the click handler of the button that checks any element marked with the `inline-label` class to see if its contents are identical to the contents of its title attribute. If that is the case, we set the contents to the empty string, as this indicates that the user hasn't filled in anything in this element and we do not want to store the text of the inline label as the content of our new task (more about this in the section on tooltips). Note that we do nothing to prevent propagation of the click event, so if this button is of the `submit` type (and our add button is) the `submit` action will still be performed.

All elements with the `del-button` class (highlighted) are then styled with an icon of a trash can. The buttons also receive an extra click handler that will remove the disabled attribute from their siblings (the input fields in the same form) to make sure the submit action will receive the contents even from fields that are marked as disabled.

Next, the other `<button>` elements are adorned with an appropriate icon and to any text or password `<input>` element we add a `textInput` class to mark it for the tooltip library.

In the second highlighted line, we encounter jQuery UI's datepicker widget. The datepicker widget greatly simplifies entering dates for the user and is now more or less a staple item in any web application or website that asks the user to enter a date. jQuery UI's datepicker is very straightforward to use, yet comes with a host of configuration options (all of them documented at <http://jqueryui.com/demos/datepicker/>).

We use the `dateFormat` option to configure the datepicker to store dates as YYYY-MM-DD. Datepicker has a number of predefined formats and this one happens to be an international standard as well as a suitable format to sort dates in a simple way. We also configure the datepicker to call a function when the user closes the datepicker. This function removes any `inline-label` class, preventing the newly entered date to appear in the colors associated with any inline label (as we see later, when we look at `tasklist.css`, we style the colors of any element with an `inline-label` class in a distinct way).

Earlier, we indicated that we wanted to present the list of tasks ordered by their due date. We therefore apply the `sort()` plugin from `sort.js` to all `<input>` elements with a `duedate` class. `sort()` takes two arguments. The first one is a comparison function that is passed two elements to compare. In our case, that will be `<input>` elements that contain a date in the YYYY-MM-DD format, so we can simply compare the values of these elements as text and return plus or minus one. The second argument is a function that takes no arguments and should return the element to be sorted. The input element with the due date is available as the `this` variable within this function and we use it to return the parent of the input element. This parent will be the `<form>` element that encloses it and because we represent each task as a form, we want those forms to be sorted, not just the `<input>` elements inside these forms.

The last set of actions in `tasklist.js` adds a `disabled` attribute to any `<input>` element within an element that has a `done` class and disables any done button. This will ensure that tasks marked as done cannot be altered:

Chapter3/tasklist.js

```
$(document).ready(function() {  
    $(".header").addClass("ui-widget ui-widget-header");  
  
    $(".add-button").button({  
        icons: {primary: 'ui-icon-plusthick' },  
    });  
});
```

```

        text:false}).click(function(){
            $(".inline-label").each(function(){
                if($(this).val() === $(this).attr('title')) {
                    $(this).val('');
                }
            });
        });
    $(".del-button").button(
        {icons:{primary: 'ui-icon-trash' },
        text:false}).click(function(){
            $(this).siblings("input").removeAttr("disabled");
        });
    $(".done-button").button( {icons: {primary:'ui-icon-check'},
        text:false});
    $(".logoff-button").button({icons: {primary:'ui-icon-closethick'},
        text:false});
    $(".login-button").button( {icons: {primary:'ui-icon-play'},
        text:false});
    $(".:text").addClass("textinput");
    $(".:password").addClass("textinput");
    $( ".editable-date" ).datepicker({
        dateFormat: $.datepicker.ISO_8601,
        onClose: function(dateText,datePicker){
            if(dateText != ''){$(this).removeClass("inline-label");}}
    });
    $("#items form input.duedate").sort(
        function(a,b){return $(a).val() > $(b).val() ? 1 : -1;},
        function(){ return this.parentNode; }).addClass(
            "just-sorted");

    $(".done .done-button").button( "option", "disabled", true );
    $(".done input").attr("disabled","disabled");
});

```

JavaScript: tooltip.js

tooltip.js is a bit of a misnomer as its most interesting part is not about tooltips but inline labels. Inline labels are a way to convey helpful information not by means of a hovering tooltip, but by putting text inside text input elements. This text then disappears when the user clicks the input field and starts typing. There are many implementations to be found on the web, but the most clear and concise one I found is from <http://trevordavis.net/blog/tutorial/jquery-inline-form-labels/>.

Time for action – implementing inline labels

Take a look again at the screenshot of the list of tasks:

Due date	Description	Completed
2010-08-28	work	2010-08-28
2010-08-29	more work	2010-08-29
2010-08-30	even more work	None
2010-08-31	relax	None

click for a date click to enter a description +

The highlighted parts show what we mean by inline labels. The input fields display some helpful text to indicate their use and when we click such a field, this text will disappear and we can enter our own text. If we abort the input by clicking outside the input field when we have not yet entered any text, the inline label is shown again.

What just happened?

`tooltip.js` shows a number of important concepts: First how to apply a function to each member of a selection (highlighted). In this case, we apply the function to all `<input>` elements that have a `title` attribute. Within the function passed to the `each()` method, the selected `<input>` element is available in the `this` variable. If the content of an `<input>` element is completely empty, we change its content to that of the `title` attribute and add the class `inline-label` to the `<input>` element. That way, we can style the text of an inline label differently than the regular input text if we like, for example, a bit lighter to make it stand out less.

The second concept shown is binding to the **focus** and **blur** events. When the user clicks an `<input>` element or uses the *Tab* key to navigate to it, it gains focus. We can act upon this event by passing a function to the `focus()` method. In this function, the `<input>` element that gains focus is again available in the `this` variable and we check if the content of this `<input>` element is equal to the content of its `title` attribute. If this is true, the user hasn't yet changed the content, so we empty this element by assigning an empty string to it (highlighted).

The same line shows another important concept in jQuery, that of **chaining**. Most jQuery methods (like `val()` in this example) return the selection they act upon, allowing additional methods to be applied to the same selection. Here we apply `removeClass()` to remove the `inline-label` class to show the text the user is typing in the regular font and color for this `<input>` element.

We also act on losing focus (commonly referred to as *blurring*), for example, when the user clicks outside the `<input>` element or uses the *Tab* key to navigate to another element. We therefore pass a function to the `blur()` method. This function checks whether the content of the `<input>` element is empty. If so, then the user hasn't entered anything and we insert the content of the `title` attribute again and mark the element with an `inline-label` class.

Chapter3/tooltip.js

```
$(document).ready(function() {
    $('input[title]').each(function() {
        if($(this).val() === '') {
            $(this).val($(this).attr('title'));
            $(this).addClass('inline-label');
        }
        $(this).focus(function() {
            if($(this).val() === $(this).attr('title')) {
                $(this).val('').removeClass('inline-label');
            }
        });
        $(this).blur(function() {
            if($(this).val() === '') {
                $(this).val($(this).attr('title'));
                $(this).addClass('inline-label');
            }
        });
    });
});
```

CSS: tasklist.css

Without some additional styling to tweak the layout, our tasklist application would look a bit disheveled, as seen before.

Our main challenges are aligning all columns and moving all buttons consistently to the right. All elements in our HTML markup that make up the columns are marked with a class to indicate that they belong in the left, middle, or right column. All we have to do to align these columns is to set their width based on their class (highlighted).

The largest part of the rest of `tasklist.css` is concerned with either floating elements to the right (like buttons) or to the left (containers, like the `<div>` element with the `id` attribute `content`). Most containers are not only floated to the left, but also explicitly set to a width of 100 percent to make sure they fill the element they are contained in themselves. This is not always necessary to position them correctly, but if we do not take care, the background color of the enclosing element might show if an element doesn't fill its enclosing element:

Chapter3/tasklist.css

```
input[type="text"] {
    font-size:1.1em;
    margin:0;
    border:0;
    padding:0;}

.left, .right { width: 8em; }
.middle { width: 20em;}

form {
    float:left;
    border:0;
    margin:0;
    padding:0;
    clear:both;
    width:100%; }

form.logoff{
float:right;
    border:0;
    margin:0;
    padding:0;
    clear:both;
    width:auto;
    font-size:0.5em;}

#items { float:left; clear:both; width:100%; }

.header { width:100%; }
.taskheader, .header, #content{ float:left; clear:both;}
.taskheader div { float:left; font-size:1.1em; font-weight:bold;}
.logoff-button, .done-button, .del-button, .add-button { float:right;}
.done-button, .add-button, .del-button { width: 6em; height: 1.1em; }
#content { min-width:900px;}
```

Note that our stylesheet only deals with measurements and font sizes. Any coloring is applied by the chosen jQuery UI theme. With the styles applied, the application looks a fair bit tidier:

Tasklist for : user		
Due date	Description	Completed
2010-08-28	work	2010-08-28
2010-08-29	more work	2010-08-29
2010-08-30	even more work	None
2010-08-31	relax	None
click for a date	click to enter a description	

Pop quiz – styling screen elements

1. In `tasklist.js`, we explicitly configured all buttons to show just an icon without any text. But what if we wanted to show both an icon and some text, what would we do?
2. If we didn't set the width of the form that makes up a task explicitly to 100 percent, what would the biggest disadvantage be?

Have a go hero – changing the date format of a datepicker

To display the date as ISO 8701 (or YYYY-MM-DD) isn't everybody's idea of a readable date format. For many people, the default mm/dd/yy is far more readable. How would you change `tasklist.js` to display the tasks with this default date format? Hint: it isn't enough to leave out the `dateFormat` option when calling the `datepicker()` plugin, you also need to change the comparator function to sort the tasks in a suitable manner.

For the impatient or curious readers: a sample implementation is available as `tasklist2.js` (start up `tasklist2.py` to see the effect).

Have a go hero – serving a task list from a different URL

One way to measure how reusable a piece of code is, is by using it in a situation that you did not yet have in mind when you designed it. Of course, that doesn't mean our task module should be able to function as a control application for an automobile construction plant, but what if we would like it to be part of a larger suite of applications served from the same root? Would we have to change anything?

Say we want to serve the tasklist application from the URL `/apps/task` instead of `/task`, what would we have to change?

Hint: In CherryPy, you can create a tree of URLs by assigning object instances to class variables of the object instance that is passed to the `quickstart()` method.

A possible implementation can be found in `tasklistapp.py`.

Summary

We have learned a lot in this chapter about session management and storing persistent information on the server. Specifically, we saw how to design a tasklist application and implement a logon screen. What a session is and how this allows us to work with different users at the same time and how to interact with the server, and add or delete tasks. We also learned how to make entering dates attractive and simple with jQuery UI's datepicker widget and how to style button elements and provide tooltips and inline labels to input elements.

Now that you know a little bit more about storing data on the server, you might wonder if storing information in plain files on the server filesystem is the most convenient solution. In many cases, it isn't and a database might be more suitable—which is the topic of the next chapter.

Where to buy this book

You can buy Python 3 Web Development Beginner's Guide from the Packt Publishing website: <http://www.packtpub.com/python-3-web-development-beginners-guide/book>

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information:

www.packtpub.com/python-3-web-development-beginners-guide/book