Tornado Web Framework Cheatsheet

A comprehensive cheat sheet for the Tornado web framework, covering its core concepts, modules, and usage patterns for building asynchronous web applications in Python.

Core Concepts & Application Structure

Tornado Application

The `tornado.web.Application` class is the central component. It maps request handlers to URL patterns.

Example:

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")


def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    app.listen(8888)

tornado.ioloop.IOLoop.current().start()
```

Request Handlers

`tornado.web.RequestHandler`	Base class for request handlers. Override HTTP method functions (e.g., `get()`, `post()`).
`self.write(chunk)`	Writes a chunk of output to the client. Can be called multiple times.
`self.render(template_name, **kwargs)`	Renders a template using the given context.
`self.get_argument(name, default=None, strip=True)`	Returns the value of the argument with the given name.
`self.set_header(name, value)`	Sets an HTTP header.

URL Routing

URL patterns are defined as a list of tuples: `(r"/path", HandlerClass, dict(kwargs), name)`

- `r"/path"`: Regular expression to match the URL.
- `HandlerClass`: The request handler class to use.
- `kwargs`: A dictionary of keyword arguments to pass to the handler's `initialize` method.
- `name`: Name of the route, can be used with `reverse_url()`

Example:

```
app = tornado.web.Application([
    (r"/", MainHandler),
    (r"/user/([a-zA-Z0-9]+)", UserHandler, dict(database=db)),
])
```

Asynchronous Operations and IOLoop

IOLoop Basics

The `tornado.ioloop.IOLoop` is the core of Tornado's asynchronous execution.

`IOLoop.current()` - Returns the current IOLoop instance.
`IOLoop.start()` - Starts the IOLoop, blocking until `stop()` is called.
`IOLoop.stop()` - Stops the IOLoop.

Example:

```
tornado.ioloop.IOLoop.current().start()
```

Asynchronous HTTP Client

`tornado.httpclient.AsyncHTTPClient`	Non-blocking HTTP client for making asynchronous requests.
`fetch(request, callback=None, raise_error=True)`	Performs an HTTP request. Returns a `Future` if `callback` is None, otherwise invokes the callback with the `HTTPResponse` object.

Example:

```
from tornado.httpclient import AsyncHTTPClient

async def fetch_url(url):
    http_client = AsyncHTTPClient()
    response = await http_client.fetch(url)

    print(response.body.decode())


tornado.ioloop.IOLoop.current().run_sync(lambda: fetch_url("http://www.example.com"))
```

Futures and Coroutines

Tornado uses `Futures` to represent the eventual result of an asynchronous operation. `@tornado.gen.coroutine` decorator allows you to write asynchronous code using `yield` (pre-Python 3.5) or `async/await` (Python 3.5+).

Example (using `async/await`):

```
import tornado.gen
from tornado.httpclient import AsyncHTTPClient

async def get_example_html():
    http_client = AsyncHTTPClient()
    response = await http_client.fetch("http://www.example.com")
    return response.body.decode()
```

Templates and UI Modules

Template Rendering

Tornado uses its own template language, which is similar to Django's.
Templates are rendered using `self.render(template_name, **kwargs)` in a request handler.

Example:

```
class MyHandler(tornado.web.RequestHandler):
    def get(self):
        items = ["Item 1", "Item 2", "Item 3"]
        self.render("mytemplate.html", items=items)
```

Template files are typically located in a `templates` directory.

Template Syntax

`{{ ... }}`	Escapes and outputs the result.
`{% ... %}`	Template directives (e.g., `if`, `for`).
`{# ... #}`	Comments.
`{{! ... }}`	Unescaped output.

UI Modules

UI Modules are reusable components that render parts of a page. They can include CSS and JavaScript.

Example:

```
class MyModule(tornado.web.UIModule):
    def render(self, *args, **kwargs):
        return "<div>Hello from MyModule</div>"

# In template:
{# module MyModule() #}
```

Security and Deployment

Security Considerations

Always sanitize user input to prevent Cross-Site Scripting (XSS) and SQL Injection attacks.
Use HTTPS to encrypt communication between the client and server.
Protect against Cross-Site Request Forgery (CSRF) attacks by using `xsrf_form_html()` in your templates and enabling `xsrf_cookies` in your application settings.

Authentication

`tornado.web.authenticated`	Decorator to require authentication for a handler. Redirects to `login_url` if the user is not authenticated.
`get_current_user()`	Override this method in your handler to determine the current user. It should return a user object or None.

Example:

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        user_id = self.get_secure_cookie("user_id")
        if user_id:
            return self.db.get("SELECT * FROM users WHERE id = %s", int(user_id))
        return None

class MainHandler(BaseHandler):

@tornado.web.authenticated
    def get(self):

self.render("main.html", user=self.current_user)
```

Deployment

Tornado applications can be deployed using various methods:

- **Standalone:** Using `python app.py` (suitable for development).
- **Reverse Proxy:** Using a reverse proxy like Nginx or Apache (recommended for production).
- **Process Manager:** Using a process manager like Supervisor or systemd to ensure the application restarts automatically if it crashes.

For production, use a reverse proxy to handle SSL termination, static file serving, and load balancing.