The `webapp` package is what lets your Meteor app serve content to a web browser. It is included in the `meteor-base` set of packages that is automatically added when you run `meteor create`. You can easily build a Meteor app without it - for example if you wanted to make a command-line tool that still used the Meteor package system and DDP.

This package also allows you to add handlers for HTTP requests. This lets other services access your app's data through an HTTP API, allowing it to easily interoperate with tools and frameworks that don't yet support DDP.

`webapp` exposes the connect API for handling requests through `WebApp.connectHandlers`. Here's an example that will let you handle a specific URL:

```
// Listen to incoming HTTP requests (can only be used on the server).
WebApp.connectHandlers.use('/hello', (req, res, next) => {
  res.writeHead(200);
  res.end(`Hello world from: ${Meteor.release}`);
});
```

{% apibox "WebApp.connectHandlers" %} {% apibox "connectHandlersCallback(req, res, next)" %}

Serving a Static Landing Page

One of the really cool things you can do with WebApp is serve static HTML for a landing page where TTFB (time to first byte) is of utmost importance.

The Bundle Visualizer and Dynamic Imports are great tools to help you minimize initial page load times. But sometimes you just need to skinny down your initial page load to bare metal.

The good news is that WebApp makes this is really easy to do.

Step one is to create a your static HTML file and place it in the *private* folder at the root of your application.

Here's a sample *index.html* you might use to get started:

```
<head>
    <title>Fast Landing Page</title>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0 user-
scalable=no" />
    <link rel="stylesheet" href="path to your style sheet etc">
</head>

    <body>
        <!-- your content -->
    </body>

    <script>

    // any functions you need to support your landing page

    </script>

</html>
```

Then using the connectHandlers method described above serve up your static HTML on app-root/ page load as shown below.

```
/* global WebApp Assets */
import crypto from 'crypto'
import connectRoute from 'connect-route'

WebApp.connectHandlers.use(connectRoute(function (router) {
    router.get('/', function (req, res, next) {
        const buf = Assets.getText('index.html')

        if (buf.length > 0) {
            const eTag = crypto.createHash('md5').update(buf).digest('hex')

            if (req.headers['if-none-match'] === eTag) {
                res.writeHead(304, 'Not Modified')
                return res.end()
            }

            res.writeHead(200, {
                ETag: eTag,
                'Content-Type': 'text/html'
            })

            return res.end(buf);
        }

        return res.end('<html><body>Index page not found!</body></html>')
    })
}))
```

There are a couple things to think about with this approach.

We're reading the contents of index.html using the Assets module that makes it really easy to read files out of the *private* root folder.

We're using the connect-route NPM package to simplify WebApp route processing. But you can use any package you want to understand what is being requested.

And finally, if you decide to use this technique you'll want to make sure you understand how conflicting client side routing will affect user experience.

Dynamic Runtime Configuration

In some cases it is valuable to be able to control the **meteor_runtime_config** variable that initializes Meteor at runtime.

Example

There are occasions when a single Meteor server would like to serve multiple cordova applications that each have a unique `ROOT_URL`. But there are 2 problems:

1. The Meteor server can only be configured to serve a single `ROOT_URL`.
2. The `cordova` applications are build time configured with a specific `ROOT_URL`.

These 2 conditions break `autoupdate` for the cordova applications. `cordova-plugin-meteor-webapp` will fail the update if the `ROOT_URL` from the server does not match the build time configured `ROOT_URL` of the cordova application.

To remedy this problem `webapp` has a hook for dynamically configuring `__meteor_runtime_config__` on the server.

Dynamic Runtime Configuration Hook

```
WebApp.addRuntimeConfigHook(({arch, request, encodedCurrentConfig, updated}) => {
 // check the request to see if this is a request that requires
 // modifying the runtime configuration
  if(request.headers.domain === 'calling.domain') {
    // make changes to the config for this domain
    // decode the current runtime config string into an object
    const config = WebApp.decodeRuntimeConfig(current);
    // make your changes
    config.newVar = 'some value';
    config.oldVar = 'new value';
    // encode the modified object to the runtime config string
    // and return it
    return WebApp.encodeRuntimeConfig(config);
  }
  // Not modifying other domains so return undefined
  return undefined;
})
```

{% apibox "WebApp.addRuntimeConfigHook" %} {% apibox "addRuntimeConfigHookCallback(options)" %}

Additionally, 2 helper functions are available to decode the runtime config string and encode the runtime config object.

{% apibox "WebApp.decodeRuntimeConfig" %} {% apibox "WebApp.encodeRuntimeConfig" %}

Updated Runtime Configuration Hook

```
const autoupdateCache;
// Get a notification when the runtime configuration is updated
// for each arch
WebApp.addUpdatedNotifyHook(({arch, manifest, runtimeConfig}) => {
  // Example, see if runtimeConfig.autoupdate has changed and if so
  // do something
  if(!_.isEqual(autoupdateCache, runtimeConfig.autoupdate)) {
    autoupdateCache = runtimeConfig.autoupdate;
    // do something...
  }
})
```

{% apibox "WebApp.addUpdatedNotifyHook" %} {% apibox "addUpdatedNotifyHookCallback(options)" %}