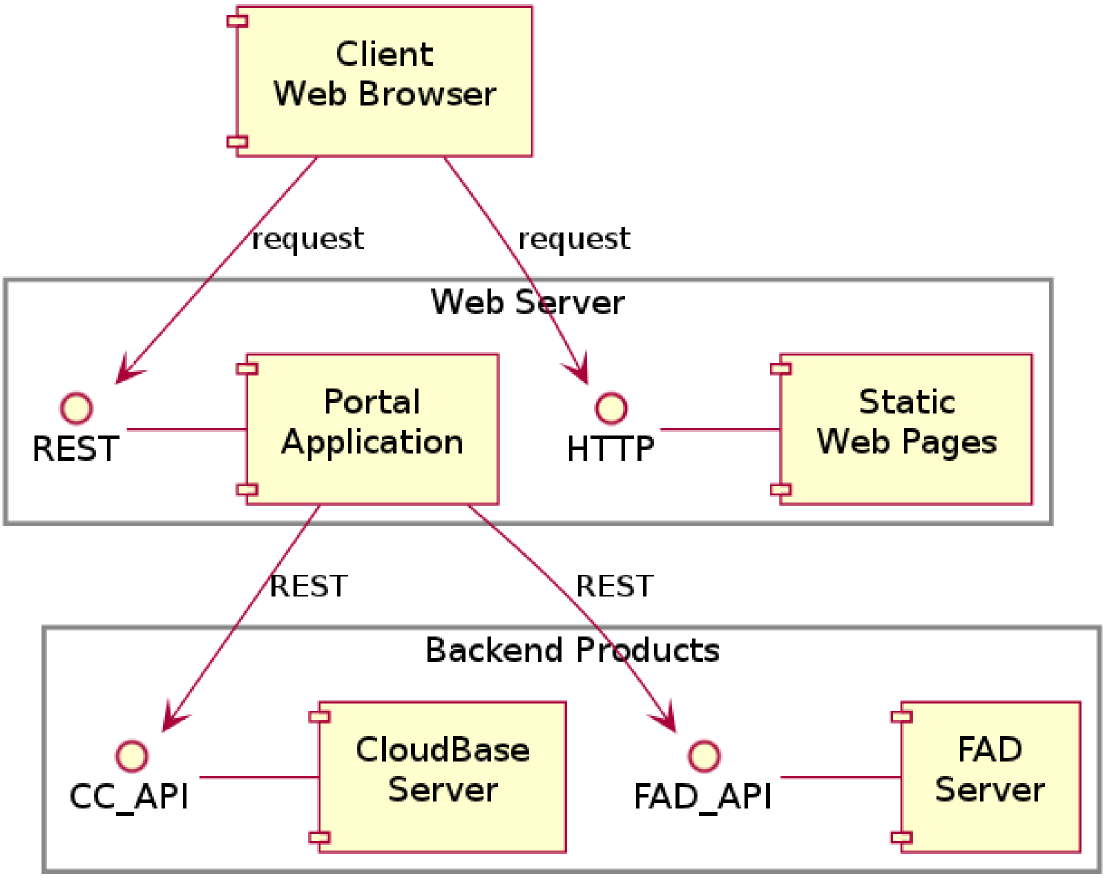
CloudPortal Architecture

Architectural Goals

CloudPortal, while a *component product* of the CloudEco product, is ***the** user experience*. As such, it needs to be dynamic and responsive. The target audience is both high-level, enterprise developers and their managers who want to view health and status of the deployed applications.

The CloudPortal components will be deployed *into* our CloudEco product system (that's right, we'll "eat our own dog food").

The CloudPortal is divided into two primary sections, the Client Application and a Server Portal Component.



Architecture Overview

The UML Component Diagram (above) is a very *high level* overview of architectural components and interfaces that make up CloudPortal:

- CloudPortal Client will be written as a dynamic web application using HTML5 technologies.
- The **ClientAPI** is a REST API customized to make the Client as simple as possible.
- CloudPortal Server answers requests from the Client, and creates new REST calls to the backend Controllers using their exposed API.
- CloudBase Controller is the primarily interface the Server talks to for information about a user's applications and services.
- The `CC_API`, at first, will be the REST API of [CloudFoundry]'s Cloud Controller. This API will evolve over time to meet our needs.

- The FAD Server is a small server that exposes a REST interface (`FAD_API`) for controlling and monitoring deployments into our cloud.
- The `FAD_API` is a REST-based API that we will develop for connecting to the CloudPortal.

Note: We will have many back-end servers for things like monitoring, health and performance information about the systems in the CloudEco installation.

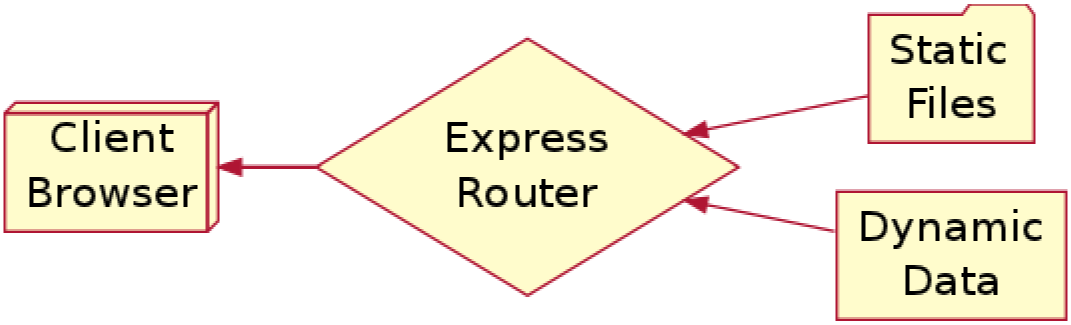
Client Application

The CloudPortal Client will be written in JavaScript using HTML5 technologies and the jQuery library. This requires the **Client Web Browser** to be a *modern* browser capable of AJAX and other dynamic JavaScript features.

The client application will be deployed to the browser from the **Web Server** as a series of *static files*, including:

- HTML Files
- CSS Stylesheets
- JavaScript files
- Images
- HTML Templates

Even though in the original component diagram (below) it appears that the client is talking to two different services, the Client will really only communicate to a single Server, but using two *types* of URLs that refer to static files and dynamic data.



The following are some example URL routes that should demonstrate this:

Static Files	Dynamic Data
`/index.html`	`/user`
`/js/jquery.js`	`/user/528392355`
`/css/styles.css`	`/user/528392355/application/4294438`

Note: The URL request for `/` will redirect to `index.html`, as this will be the start of the entire client application.

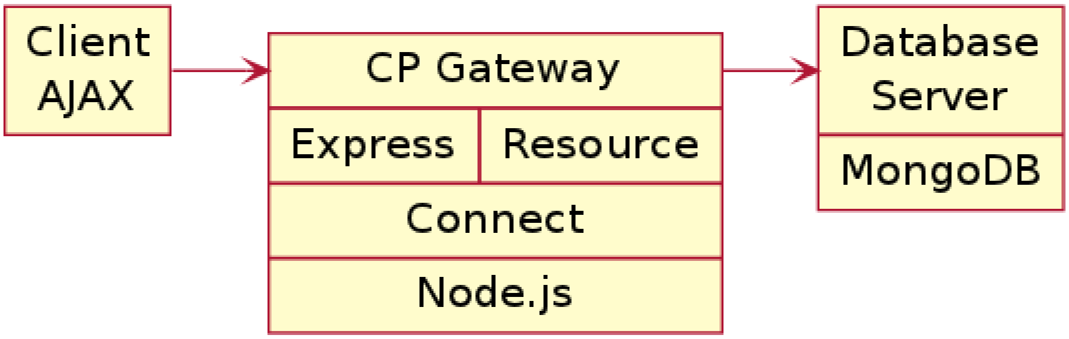
From the standpoint of the server, all client files are *static*. That is, the server will not spend any processing cycles rendering views. Instead, the client will download both the data (through the REST API) and a template and process the view on the client. We will use the FuzzyToast library to make this process easier.

Once the client parts have been downloaded and executed, the client application will call back to the **Web Server** with a series of REST requests over the HTTP protocol. The response will contain JSON-formatted data (this includes error message details).

Server Portal

As shown in the diagram below, the Server components are split into two separate servers:

- [CloudPortal Gateway] is the only interface to the client. It answers front-end REST requests by making back-end REST requests.
- [CloudPortal Database] is a server that connects to a MongoDB database and stores data specific to the CloudPortal system.



These servers will be written in JavaScript (the same language as the Client, but using a different set up technologies:

- Express is a high-level framework that allows us to create the REST API directly as a collection of functions. We will also use the express-resource extension.
- Connect gives Express its abilities to process Cookies, HTTPS and other mid-level features.
- Node.js is a low-level, asynchronous framework for creating web applications on the server. It uses Google's V8 engine.

Note: We might want to look at ql.io for the Gateway, as this technology maps multiple *incoming* REST calls to a single *outgoing* REST API. Its database-like `SELECT...FROM...WHERE` syntax is a bit odd, though.