2012

A Web Application in REST: The design, implementation, and evaluation of a web application based on REpresentational State Transfer

William O'Brien
Colby College

A Web Application in REST

By William O'Brien

A Web Application in REST

The design, implementation, and evaluation of a web application

*based on **RE**presentational **S**tate **T**ransfer.*

2012 Honors Project

William O'Brien

Advisor: Dale Skrien

Department of Computer Science

Colby College

Waterville, ME

Table of Contents

Abstract 3

Introduction 4

Background 10

SOAP 11

REST 12

Resources 15

Constraints 20

Criticisms of REST 26

Problems this Project Addresses 30

Implementation 32

Structure overview 32

Implementation Details 37

The Client-Side 51

Registration 51

Mobile 54

Browser 55

Evaluation 56

Summary 59

Glossary 60

References 64

Index 66

Appendix A - JSON Schema Validator on NodeJS 68

Appendix B - Dojo Form Extension 70

Abstract

It is no secret the Internet has evolved at an alarming rate since its public debut in the early 90s. What began as a simple way for users to publish files has evolved into a platform for impressively complex interactive applications.

Now, applications once restricted to a single operating system are made available to all through advancements in browser technology and libraries. For a long time, Adobe has offered Photoshop as a far from trivial image editing program for around $100 to the Windows and Mac platforms. Recently, a web application emulating Photoshop's features has been made freely available to anyone with an Internet connection and a web browser, regardless of operating system.

Not only are many of today's websites complex, they also span much further than they were ever previously capable of. Websites such as Facebook extend tentacles across hundreds of thousands of websites in order to allow websites to update a user's status or to 'like' content.

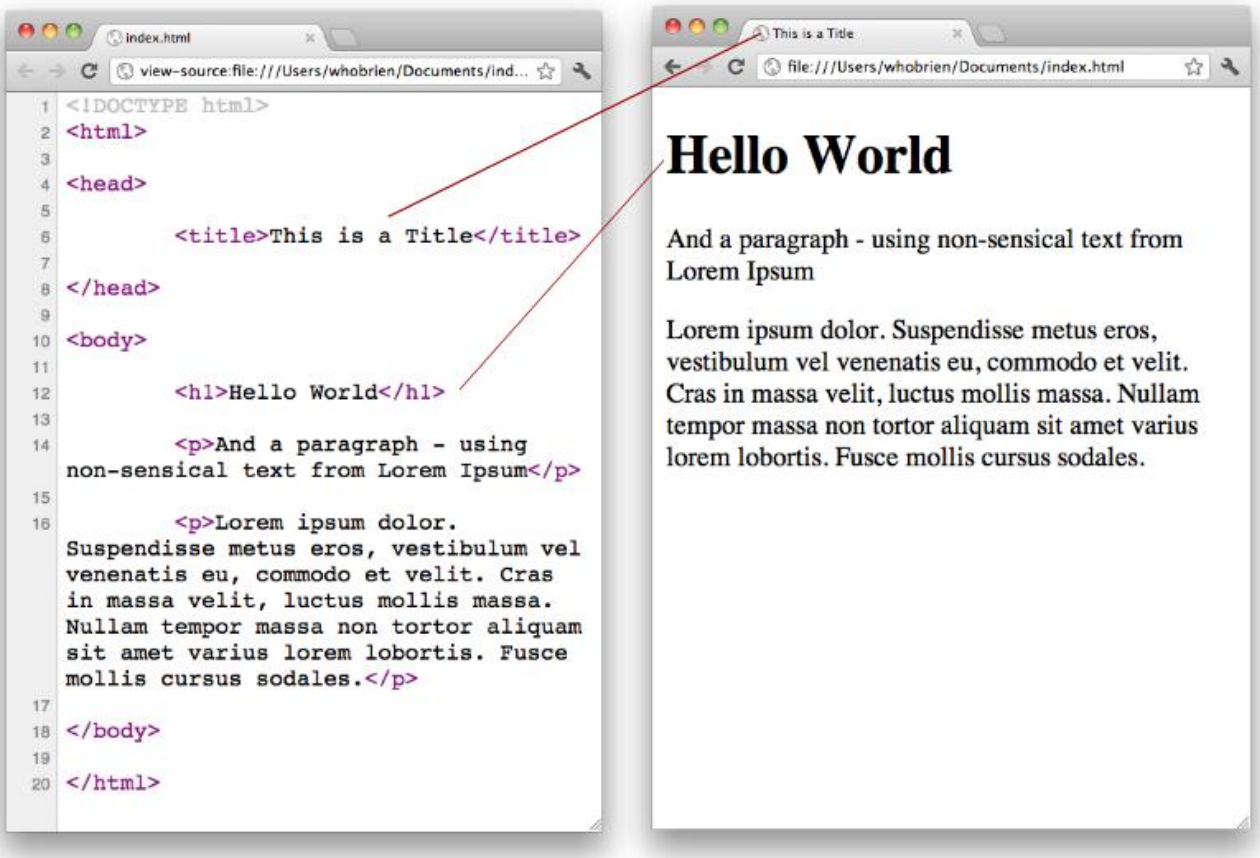
With rapidly increasing complexity and adoption, it is more important than ever for developers to ensure longevity and scalability in their applications. The style in which developers write their applications is largely responsible for eliciting and maintaining these qualities. There is much written on tried and true architectural styles for network based applications and their employment is paramount in handling the dynamic nature of the web.

This project, Blockstep, attempts to implement the golden standard of web architecture, Representational State Transfer (REST). Blockstep will employ a new, standards-track data format to be incorporated into the machine-readable part of this application.

Introduction

These days it is almost instinctual to type in a web address into a browser's address bar and expect the desired site to appear shortly after the stroke of the enter key. What happens, though, between the tap of the key and viewing the page? There are rumors of HTML, XML, CSS, and other strings of characters quaffed forth from the strange telephone like plug in the back of the computer. Occasionally, one wanders into the 'source' of the web page revealed by the browser to see a slew of `<brackets>` and http:// links. What is hidden from us, and thankfully so, is all the work the browser does to present this essay of gibberish in a coherent form.

As many sites do, we will begin with a simple web page (see Figure 1) consisting of a document type declaration, `<!DOCTYPE html>` which will indicate to a browser that the following document should be rendered according to the HTML5 specification . Inside this `<html>` tag will be the contents and meta-data about the page; this page will include a `<title>` tag and some text. When the browser receives this document it will sort these tags and render them appropriately—notice the title goes to the top of the browser window, the content is displayed in the main section, and the `<h1>` heading tag is bold and enlarged.



This document sits on some remote computer waiting to be requested. Since the job of this computer is to sit on a network connection and run an application that will listen for and respond to requests, it is called a 'server'. The application will 'serve' the documents requested across the network.

A user wishes to view the page enters the document's URL into the address bar and the browser generates a simple request to the server that is hosting the file:

```
GET /index.html HTTP/1.1
Host: example.domain.com
Accept: text/html
```

The server receives the request[1] and responds with the file indicated by the request. A response code `200` means the request was successful:

[1] For simplicity, only the most important request and response headers are included in this example.

```
HTTP/1.1 200 OK
Content-Length: 677
Content-Type: text/html

<!DOCTYPE html>
<html>...</html>
```
[2]

After the browser receives the file, it will strip the `<tags>` from the document and render the results in accordance with the HTML5 specification we indicated in our `<!DOCTYPE>` tag.

If an administrator of the web site wants more than a bare-bones web document they could add some styling markup in the form of Cascading Style Sheets (CSS) to make things look aesthetically pleasing (or terrifying). To do this, they include a link in the HTML document to a CSS document. Here is an exchange between a client[3], and the server for a web document with a link to a CSS document.

Request:

Headers

```
GET /index-links.html
Host: example.domain.com
Accept: text/html
```

Body

None

Response:

Headers

```
HTTP/1.1 200 OK
Content-Length: 757
Content-Type: text/html
```

Body

```
<!DOCTYPE html>
<html>
      <head>
            <link type="text/css" rel="stylesheet" href="/style.css" />
      </head>
      <body>...</body>
</html>
```

Now that there is a link to a `text/css` document in the HTML page, the browser will automatically create a `GET` request to retrieve the sheet.

---

[2] Some markup stripped for clarity

[3] The user's browser

Request:

Headers

```
GET /style.css HTTP/1.1
Host: example.domain.com
Accept: text/css
```

Body

None

Response:

Headers

```
HTTP/1.1 200 OK
Content-Length: 168
Content-Type: text/css
```

Body

```
h1 {
    color: green;
    background: #CCCCCC;
}
section > * {
    margin: 0 auto 2em auto;
    width: 300px;
}
```

Notice that the browser has recognized the link as a link to a CSS document and has requested in the `Accept` header a media type of `text/css`. The server will respond appropriately with a corresponding `Content-Type` of `text/css`. Upon receipt of the response, the browser will change the way it has rendered the HTML according to the markup in the style sheet.

If an image had been included in either the HTML or CSS documents, a third `GET` request would be made to the server with a request header of `Accept: image/*` indicating the browser is expecting a picture, but it is not sure what type of image. The server's response would include a header indicating what type of image it is: `Content-Type: image/png`.

Web administrators and users typically require much more than these simple content requests. Users want to be able to update the website from their browser and administrators want to collect data from forms as well as display dynamic elements like the date and time or number of hits. Both users and administrators want animations on the page as might be characteristic of native applications. Until this

point, this example has only included static markup and images – only straight HTML, CSS, and images have been received and rendered. Dynamic features can be added to these static documents partially through server-side programming; the documents served to browsers can be generated differently for each request. Languages like PHP make the addition of dynamic content as easy as entering script fragments directly into the server-side document to be interpreted on the fly. Additionally, hypertext documents may include links to source code to be run in the browser. ECMAScript[4] for example, better known as JavaScript, can be used to support user interaction by manipulating the page displayed to users.

Over the last decade, the pages delivered to the browser from the server have become more comprehensive and have recently started to resemble native applications. The requests made to the server after the initial page-load is smaller, asynchronous queries for data fragments to update parts of a page rather than navigating to an entirely new page. These exchanges are categorized as Ajax[5] requests. These simple interactions with the server allow the browser to manipulate server-side application data through designated URIs. Through the user of JavaScript, all the logic for displaying and managing application state can be done in the browser instead of on the server[6]. The use of Ajax has reduced the number of full-page requests resulting in the perception by users that pages are loading faster.

While it may seem that pages load faster, the explosive popularity of the web has made it challenging to keep track of all the clients being served and respond to them in a reasonable amount of time. Further, web page designers do not want to be limited to sending messages exclusively to the origin server. They want to, for example, send messages to Facebook to add a notification that a page has been 'liked', or they want to update Twitter with the title of the user's last read article.

---

[4] For now, other client-side languages and executables are intentionally omitted

[5] At one point Ajax was the acronym AJAX, Asynchronous JavaScript and XML. This acronym has been deprecated in favor of the label Ajax to refer to the asynchronous style of interaction without implying specific implementation forms. Ajax can be used with protocols other than HTTP and is often used with non-XML data formats like JSON.

[6] For those familiar with an MVC architecture, think of this as moving the VC, and to some extent M, parts to the browser.

In order to service their own applications, as well as the needs of other applications, companies like Facebook, Twitter, Amazon, Netflix, etc. have developed special public services to accept and respond to these Ajax requests. Other companies like Google provide services for these browser-based programs that allow an application to outsource computationally intensive or complicated algorithms to a dedicated service—*see the Google Prediction API*. The design and implementation of these types of services is the subject of this paper.

REST, as an architectural style, comes standard for most document-based web servers. When creating a service to generate custom responses for non-human clients, maintaining a RESTful architecture is not trivial. This project, Blockstep, is an attempt at implementing one such service. The challenge in developing a RESTful web service is that the responses delivered must be interpreted by client code rather than an intelligent human being. The client application must therefore have some understanding of the structure of requests and responses so it can display options to a user accordingly. In this case, the application consuming the service decides how to display options rather than the service, as would be the case in an HTML exchange. In order to interpret the results, the application consuming the service must have some concept of what format the data coming across the wire is in. This project applies a new standards-track data format called JSON-schema to make the current format of data, JSON, less ambiguous to client application developers.

The rest of this paper assumes the reader has a basic understanding of HTTP, client-server interaction, and of browser technology.

Background

Since its inception, the Internet has made a dramatic transition from static hypertext pages to intelligent dynamic documents. The HTTP protocol, on which the World Wide Web is based, cites at its core hypertext documents – documents that reference other documents through links. The original hypertext documents were static. Retrieving a document would be the same any time and any place until the server administrator edited the contents of the file being served. As server technology advanced, these hypertext documents began to include dynamic elements. For example, a document could include the date a specific instance of a page was served or even content posted by clients of the server. To expand the dynamic nature of a page, web curators began to embed runnable code in their hypertext documents. Over the years this code matured and began to leverage asynchronous calls to servers to update the state of the page as well as the state of information on the server. Code embedded in the hypertext documents, received by the client, constructs HTTP messages to send to other servers in order to affect the state of the application or other web based applications[7]. Corporate sites, such as Twitter, Google, and Facebook, open their hypertext-based programing interfaces to independent programmers on the web. A need arose to identify and standardize the best practices of constructing these massively distributed applications on the Internet. It is unreasonable for each service to require use of a new site-specific protocol.

Two widely accepted approaches to web services have emerged to address the need for consistency across the application programming interfaces (APIs). REST and SOAP are the two prevailing approaches to constructing these services. While many developers swear by one or the other, each has its place, although as will be explained, the more RESTful a service is, the better.

[7] Websites that serve hypertext documents that allow a user to affect the state of the site will be referred to as web applications. Often the documents served by these sites have code embedded in them that facilitates state changes and enhance interactivity.

SOAP

SOAP is an XML-based protocol for sending and receiving messages from application servers. The benefits offered by SOAP are generally of interest to enterprise-type applications, especially those interested in accessing multiple web services in one action or those that deal with financial data.

The mail must go through

Unlike the REST architecture, SOAP has built-in procedures for retrying unsuccessful message delivery. This is important for applications that rely on transaction integrity – everything completes or nothing completes. Think banking transactions between two disparate banks; money must only be deposited at one bank if it is successfully withdrawn from the other.

There's an app for that

The maturity of SOAP has yielded powerful tools for implementing both the server and client-side code. This is important because the structure of the messages is much more complex than one might find in a comparable REST architecture. In the words of Amit Asaravala, “a four or five-digit stock quote in a SOAP response could require more than 10 times as many bytes as would the same response in REST.” (Asaravala) If there were not tools to generate these messages and contracts, construction of the messages would be burdensome and time-consuming.

It's in the contract

The services provided by SOAP based APIs conform to a strict contract of service. The service descriptor might dictate: if a message is received in exactly this form, this will be the form of the result. Else, x or y will be the error codes.

These computer-readable service contracts can be discovered by scanning a network. Code can be generated from these descriptors with use of the aforementioned tools to interface with a service.

These characteristics make SOAP-based web services great for mission-critical applications that back large corporations. There is no denying SOAP has its place, but rarely in public facing web applications do these characteristics outweigh the benefits afforded by a RESTful service architecture.

It is important to note, SOAP is a protocol for invoking services across the Internet, frequently using HTTP as its underlying transport protocol. REST is an architectural style that SOAP violates. The two approaches to web services are exclusive of one another. Both are used to accomplish similar tasks but one should keep in mind that this is not a comparison of apples to apples.

REST

When a web service[8] is said to be RESTful, it and the rest of the application[9] conforms to the architectural constraints outlined by Roy Fielding in his doctoral dissertation. (Fielding) If these constraints are followed, the application will be able to leverage all the benefits that HTTP was originally designed to bring to massively distributed applications. The architecture is intended to promote several software qualities, most important of which are user perceived performance, scalability, and extensibility. The focus is on the application at a network level although constraints extend into the software implementations. Representational State Transfer (REST) attempts to elicit these software qualities by making the following constraints on an application.

Statelessness of client-server interactions

A client-server relationship is generally a staple of network-based applications. It requires a separation of concerns. A client creates requests and a server reacts to them. To be stateless, a

[8] The API provided by the server to allow client code to affect the application's state.

[9] The application includes all the representations sent to and rendered by the client (a browser), the software running on the server or servers, and the messaging between these components.

request to a server, and the subsequent response, must be independent of all other requests. The server has no reliance on the context of the request among others – e.g. session data.

Limited set of actions

Facilitated by HTTP, only a small set of verbs are permitted in communication between the clients and server. Each verb has well defined properties which will be covered later. Operations are limited to create, read, update, and destroy (CRUD).

Cacheable

Responses to read actions must specify how they are to be treated by caches. Developers should strive to make extensive use of caching when possible as improves scalability significantly. If caching headers of a previously retrieved request/response indicate a saved response is not yet stale, and a second identical request is received by a client, server, or intermediary node[10], the saved response can be returned immediately rather than passing the query towards the server for processing.

Uniform interface

Given the limited set of verbs permitted, each resource, identified by a uniform resource identifier (URI)[11], is expected and required to treat each verb as it is defined. A resource may choose not to allow some verbs to be executed but it may not reinterpret the meaning of the verb.

[10] An intermediary node is any node that lies between the client and the server. E.g. the routers of your Internet service provider that allow you to connect to the web.

[11] E.g. 'http://example.domain.com/this/is/a/uri'

Limited representations of state

The format of a response's payload is constrained to a particular content-type understood by both the client and the server. These are declared by a media type[12]. A server may be configured to serve pages of different media types but it should not be expected that browsers will understand the responses.

Layered System

A layered system is an architectural constraint that expands on the client-server requirement. This constraint requires applications to be organized into logical divisions. For example, a web application might have a view layer, a business logic layer, a persistence layer, etc. The intent is to decouple each layer so it may be swapped out for a separate implementation without having to rewrite the whole program.

The reason these constraints are so critical for large service providers is the massive scale of the Internet. Even with the ever growing processing power of modern computers, users are logging on more often and requesting more content. Debuting websites that make it to the front page of Digg, Reddit, or StumbleUpon need to be prepared to scale to handle a thousand-fold increase in traffic in a matter of minutes to capture a potentially debilitating flood of requests. Maintaining a RESTful service architecture would make that transition as easy as clicking a few links in the cloud to start a few additional servers. The flood of requests may be distributed to these new servers. Each of these constraints will be covered in more detail to illustrate how this is made possible.

[12] A media type is a string included in a request or response to indicate the form of the appended document. Most of the commonly accepted media types were originally defined as multipurpose Internet mail extensions (MIME types) defined in RFC 2046.

Before justifying these constraints, a basic understanding of how these constraints are realized will provide a framework for discussion.

Resources

Dr. Fielding proposes a number of best practices in constructing and maintaining a uniform interface. The concept of resources is one principle that is a cornerstone of creating a RESTful web service.

A resource is any addressable noun. It represents a concept. For example, Blockstep has a users resource at `/api/users`. Conceptually, this resource represents the group of users of the site. A `GET` request to this resource could be satisfied by a number of representations of the `/api/users` resource's state. In this case, a list of users might be returned as JSON, HTML, XML, or any medium the server wished if the request does not specify. If a browser is indifferent to the format of the response payload it would specify a header `Accept: */*`. If the record for a specific user was requested, a RESTful interface would specify which user is desired by appending an identifier to the end of the resource URI (`/api/users/49392`). The resulting response might be any of the following representations of the single user identified by the id 49392:

XML **Content-Type: text/xml, or application/xml**

```
<?xml version="1.0" encoding="UTF-8" ?>
<user id="49392"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="myUserSchema.xsd">
    <name>
        <first>Randall</first>
        <last>Monroe</last>
    </name>
    <address>
        <number>3333</number>
        <street>Coyote Hill Rd</street>
        <city>Palo Alto</city>
        <state>California</state>
        <zipcode>94304</zipcode>
    </address>
    <password>SXQncyB0aW1lIGZvciB5b3UgdG8gZmluZCBhIGhvYmJ5Li4u</password>
</user>
```

JSON **Content-Type: application/json**

```
{
    "id": 49392,
    "firstName": "Randall",
    "lastName": "Monroe",
    "address": "3333 Coyote Hill Rd, Palo Alto, CA 94304",
    "password": "SXQncyB0aW1lIGZvciB5b3UgdG8gZmluZCBhIGhvYmJ5Li4u"
}
```

HTML **Content-Type: text/html**

```
<!DOCTYPE html>
<html>
<head>
    <title>HTML User</title>
</head>
<body>
<section>
    <h1>Randall Monroe id #49392</h1>
    <p>Address: <br />3333 Coyote Hill Rd, Palo Alto, CA 94304</p>
    <p>Birthday: <br />November 29th, 1973</p>
    <p>Password Hash: <br />SXQncyB0aW1lIGZvciB5b3UgdG8gZmluZCBhIGhvYmJ5Li4u</p>
</section>
</body>
</html>
```

Image **Content-Type: image/png**

All four of these representations of `/api/users/49392` represent the same state of the user. Notice that not all representations include the same data; an image for example does not include the address, password, etc. of a user, while a JSON representation will not include an image.

If a user wants a particular representation, they will specify in the request's headers which format they would prefer. Similarly, a browser can 'negotiate' content types with the server by sending a prioritized list of media types in the `Accept` header of the request with varying specificity. `application/some.class+json`, `application/json`, `text/*`, `*/*` for example would indicate the browser preferred a specific form of JSON over regular JSON. If neither is available, send a text

representation. And if none of those are available, send any representation. The server will respond with a fitting representation if the request can be filled, otherwise an error will be returned.

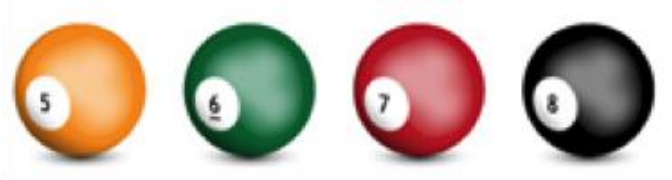
The following billiards game example will illustrate the limited set of actions available for execution on these resources.

Suppose there exists a billiards game in a pool hall somewhere on campus. A web server is set up to keep track of the game. A game is composed of 16 balls. In this example, a billiard-balls resource will be exposed at `/billiard-balls`. A representation of these balls may be retrieved by sending a `GET` request to the resource's root:

```
GET /billiard-balls
Accept: image/*
```

The response to this request is a picture of the state of the resource. The picture is the representation of state. Since all 16 balls are in the game, the picture depicts 16 balls. If it were a game of 9-ball, only 10 balls would be returned including the cue ball. Assume the author of this API is interested in making the subset of balls that are on the table available to the public at any given time. An additional path segment called table represents a subset. The full URI will be `/billiard-balls/table`.

```
GET /billiard-balls/table
Accept: image/*
```



Note that the current balls on the table are a subset. The particular sub-set returned will change over the course of a game. The same request at a different time would likely yield a different set of balls. However, the meaning of the URI has not changed, even though its state has.

The API author also wants to make available representations of individual balls. Appending its ID to the end of the parent resource can reference a ball.

```
GET /billiard-balls/3
Accept: image/*
```

In a similar way, `/billiard-balls/5` might be retrieved. Given that the 5-ball is still on the table, it might also be addressed by `/billiard-balls/table/5`. Once it is removed from the table however a `404 - Not Found` will be returned for requests to the latter URI.

Now that the resources of the billiards game have been established, they can be manipulated. Three verbs can be executed to modify state. The `GET` command, which we use to retrieve representations of the resource, is not included since it does not modify the state of a resource. The three verbs are `POST`, `PUT`, and `DELETE`, which add, update, and remove respectively. In `POST`ing and `PUT`ing, a representation of the resource being created or updated is generally required in the body of the request.

The following table indicates what each of these calls would do.

	GET	PUT	POST	DELETE
/billiard-balls	Retrieve a list of billiard balls in the game	Update the game to include only the balls contained in the body of this request. (Perhaps in switching to a game of 9-ball)	Create a single new ball and add it to the set of balls in the game	Remove all the balls from game
/billiard-balls/3	Retrieve a representation of the 3 ball	Update the representation of the 3 ball with the representation in the body of this request	Treat the 3 ball as a set of balls and create a new ball within what is now a set of balls at location [13]/billiard-balls/3	Remove the number 3 ball from the game

Many sites on the web only use GET and POST, while ignoring PUT and DELETE. Sites that do this often bastardize the use of GET and POST. GET especially, is not intended to change the state of the application. GET is considered a *safe* method because it only retrieves, it does not modify.[14]

GET, as well as PUT and DELETE are also *idempotent*. One should expect a single request using an idempotent method to have the same effect as any number of identical requests sent at the same time. For example, sending a DELETE request to our user, Randall, at /api/users/49392 should have the same effect and response that 5 more identical requests have. The status code of the nth response should not return an error if the first response was successful. After each of the DELETE requests, as desired, the user will not exist.

POST is an example of a method that is not idempotent. A POST request to the /api/users resource, in Blockstep, will create a new user. If an identical request is sent, either a 2nd new user is created or the

[13] Since this operation would not make sense in the context of this resource, it will likely be disallowed, returning instead a status code of 405 - Method not allowed.

[14] RFC 2616 Sec. 9.1

server should return an error. In this particular project, since a username must be unique, the second request returns status code `422 Unprocessable Entity` indicating the request was well formed by the payload was semantically invalid. A symptom of `POST` and `PUT` misuse are undesired form resubmissions when an impatient visitor submits a form multiple times.

The constraints imposed by REST, outlined previously, offer a number of important benefits and deserve a more thorough explanation.

Constraints

Statelessness of Client-Server Interactions

The order and context of the requests received previously by the server must be irrelevant in processing the current request. It would require far too much overhead if the server had to keep track. Given the layered nature of services, each request made by a user might be filled by a different application server. A need to share information about the state of the client's request would require large amounts of data to be shared between the servers limiting an application's ability to scale. Additionally, concurrency issues might be introduced. A server should process a request without needing any more information to fulfil the request than what is included in the HTTP message.

One might wonder how this would work with, for example, an application that requires a multi-page form. How will the form be saved between requests before its final submission? The solution to this, as well as almost every other problem posed to REST, is to create a resource and persist it to long-term storage. The user begins with a `POST` to the resource indicating the creation of a new form-backing object. After the user completes the first form, an update request will be sent to the resource with the fields that were filled in, partially updating the entire form-backing resource. The user will then be directed to the next form that will again modify the same resource. In this way, it does not matter which of the many forms are

submitted first or which server processed the request for each form. Once the resource has all the required information, the resource can be submitted to whichever resource is responsible for processing. The interactions remain stateless because the server does not track the order or context of requests; it simply retrieves and updates the form-backing resource from a database.

Limited Set of Actions

Limiting the set of available actions is important for maintaining a uniform interface. Barring an extensive and likely futile campaign to the Internet Engineering Task Force (IETF), one cannot modify the HTTP protocol to include additional verbs. To introduce additional verbs into your application would require a `GET`, `PUT`, `POST`, or `DELETE` request to an existing resource. If that resource is a verb, e.g. `/api/users/chargemoney/` then the meaning of a request is ambiguous. `GET /api/users/assessfee/` could mean get money from a user, get the algorithm for the fee assessment, or any number of things which sound as if they might have undesired consequences. In order to avoid this confusion, only the create, read, update, and destroy methods are permitted.[15] That way, a consumer of the service can reliably predict what a call to a method will do without wading through extensive documentation. All services must have a uniform interface. With this predictability, it remains safe to do things like caching resources retrieved.

Cacheable

Since the responses from servers maintain the idempotent and safe properties, intermediaries can read the headers and cache the responses to `GET` requests in the way specified by the response headers. Caching is an important part of scaling an application as it allows read-only requests to be filled without ever hitting the application server. The client, server, or intermediaries may return a response to a previously made read request if it's saved value has not expired. The difference in response times between

[15] It is important to note that while the HTTP methods map well to these operations, in order to decouple the architecture from any particular transport protocol, a distinction between the two must be made. REST is an architectural style and is therefore not coupled with a particular protocol. HTTP does so happen to be a particularly good fit since the two were developed in parallel.

a cached resource and a freshly served resource can be several orders of magnitude - much like retrieving data from RAM rather than a hard drive. The server dictates the length of time representations may be saved in a cache. If required, a client may request a non-cached resource. If `GET` requests were permitted to affect the state of the application, caches would interfere with the realization of those intended effects. The application server might never receive the `GET` request because a cached resource was instead returned; meanwhile, the client might believe its request has succeeded in modifying state.

To put the value of caching in perspective, suppose a student is assigned an essay that requires handwritten pages about some topic. After the paper is finished the student hands it in. The teaching assistant also requires a handwritten copy. Rather than spend a day writing out a duplicate essay by hand, the student makes a photocopy of the one he turned in to his teacher. The time taken is an order of magnitude smaller than the time copying by hand would have taken. Same goes for cached request/responses. The student need not be the one to keep the copy. If the TA passed the request through the professor, the professor needn't bother the student at all but rather make a copy on the spot.

Without re-entering the SOAP vs. REST discussion it is worth noting that SOAP requests are not cacheable since they are sent using the POST method.

Uniform Interface

The uniform interface of RESTful web services makes them easy to predict. Essentially, collections of resources are manipulated and read using the four main methods, CRUD (`GET`, `PUT`, `POST`, and `DELETE` in HTTP). This makes it easy to design tools that can use RESTful APIs without knowing anything about how future APIs or unknown APIs are implemented. Many JavaScript libraries provide tools to make Ajax calls over HTTP. They have gone even further create library tools for retrieving and updating resources that implement the uniform interface. It is safe to write these libraries because REST assures the users that the interfaces will be uniform.

From a holistic standpoint, uniform interfaces allow each of the services provided on the web to evolve independently from all other services. Since the behavioral expectation of a standard interface does not change, implementations and specific representations may evolve without worry that consumers of the services are coupled with anything but the uniform interface.

This constraint is a stumbling block for many service providers and consumers. Assumptions are made about the contents of the returned data or location of resources that are not made known by the interface. Dr. Fielding refers to such assumptions as 'out-of-band information' and is quick to point out this violates this constraint as well as the limited representations of state.[16] To illustrate this, suppose two users, Jack and Tom, have representations at `/api/users/jack` and `/api/users/tom` respectively. A client of the API needs to retrieve a representation for a user named Mary. To assume Mary's representation can be found at `/api/users/mary` is considered out-of-band information. The server must have indicated Mary's URI in a response, such as might be received in a `GET /api/users` request. It would be entirely valid for an API developer to make the location of Mary's user `/api/users/3sadg23gba`.

Limited Representations of State

It is important to have limited representations of state because client applications are often written long before the services they consume. In order for a client application to render the representations received from servers, the response must conform to standard format. The greatest example, of course, is a web browser. It was written to render HTML, among other markup and languages. Many websites a browser renders have been written well after the browser source code but since the sites conform to a well-defined standard representation, the browser can make use of the received representation without patching to accommodate each individual website.

---

[16] "31." Web log comment. *REST APIs Must Be Hypertext-driven »*. Ed. Roy T. Fielding. 23 Oct. 2008. Web. 14 Apr. 2012. <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>.

The publication of a book illustrates limited representations of state. The same book can be written in hundreds of different languages, English, Spanish, Tagalog, Maori, Hindi, etc. Each of these languages has a definite syntax. A book can be represented in any of these languages while conveying the same concepts and ideas. It can be understood by any user (client) that can read the language. Publishers are not permitted to invent their own language because no client/reader would be able to make use of it. There are a limited number of languages in the world, so there are a limited number of representations of a book. Similarly, there are many markup languages or ways to represent data, but it is not feasible to expect a client to make use of a non-standard representation invented by a web application.

Taken together with the previous constraint, a uniform interface, it is important to note that no assumptions may be made about the representations received by the client. As hinted at earlier in reference to out-of-band information, any assumption of the structure of HTML, XML, or other representation received by the client is not permitted. As an example of fetid code, a client cannot send a request to the `/api/users` resource for XML and expect that the XML returned will be a list of users with a root node `<users>`. In the same way, one cannot request a book in Japanese with the expectation that they will receive a Japanese book about flying machines. All that can be expected is that the representation returned will be a book in Japanese, or in the case of the users, valid XML. As will be seen later, JSON schema is used in Blockstep to permit assumptions about structure of HTTP payloads.

Layered System

A layered system means that the application is divided into logical layers which are only aware of the two adjacent layers. In a client-server setup, a layer provides interfaces to the layer above and makes calls to the interfaces of the layer below. A gateway only knows about the link to an Internet service provider and the application servers. Or in a program, the controllers only know about classes in the service layer and not the data access layer used by those services. This decouples components from each other and allows a

web service to scale in a cost effective way, albeit with some added latency when one layer makes network calls across the network to the layer below.

At a hardware level, multiple devices can be used to service the same API. To a service provider, this means an Internet-scale volume of requests can be services by many cheap nodes. As demand increases, services can scale their capacity to handle requests almost linearly with cheap hardware. All that is required is another unit be added to the bottlenecked layer.

A gateway, for example, can be used for load balancing. Five servers might all run the same web application but requests are parceled out in equal proportion by a load-balancing gateway. These servers might defer authentication to a sixth piece of hardware running an authentication service. Or a firewall might sit between incoming requests and the load-balancer to reject unauthenticated requests to a certain section of the API.

Layering also opens up more opportunities for caching. There may be a cache between any components to lighten the load on any layers of service. Outside of the service hardware, the layered system allows all nodes between the hosted service and the end user to employ a cache.

Code on Demand

Code on demand, not previously mentioned, is an optional constraint. Truly, this deserves its own section as it, as will be evident later, is the crux of determining whether or not Blockstep successfully creates a RESTful application.

If an application wishes to employ functionality not provided by a client application, it must link source code to the representation. Application developers can enhance the user experience by implementing

functionality not immediately available using standard markup. HTML, for example, does not define drag and drop functionality but using JavaScript, which is permitted in HTML, and the click events provided by HTML, a developer can implement this on her own. It would not be acceptable to have the client download a browser patch implementing the desired functionality.

As the provider of the service, this executable code can make assumptions about the interface of the web application. As discussed before, under the limited representations of state constraint, typically assumptions cannot be made past the type declaration. It is because the code on demand and the API are developed in parallel that extra assumptions about structure may be made. Code on demand is not constrained to the uniform interface. If the client is using code from the service provider, then it need not worry about decoding arbitrary XML representations as it is done for them automatically by the on-demand code linked to the page. Presumably, the code will modify the client's rendered representation in some way allowing the user to navigate through the application's interface.

By following the above outlined constraints, applications will be more robust, simple, and scalable. Compliant applications will complement the massively distributed nature of the web, rather than fighting it.

Criticisms of REST

Fielding's dissertation was published in 2000. In 2004, HTTP's version 1.1 standards were published. Much of the work first laid down in Fielding's dissertation is reflected in the HTTP 1.1 standard. The widespread use of HTTPD and HTTP owe their success to their conformity to a RESTful architecture. It wasn't until 2005 that the term Ajax was coined and later still that public APIs became popular. REST-

styled APIs came about as a back-to-the-basics style of designing these APIs. Many APIs claimed use of REST and fervently promoted it. A number of criticisms of REST as an API style have arisen, several of these criticisms stem from experience with APIs that do not conform to Dr. Fielding's constraints, yet claim to be RESTful.

Machine readability

SOAP proponents tout the benefits of having a web service description language (WSDL). Machines can scan the network to find these documents and immediately know about the endpoints[17] at which an invokable method exists as well as a series of defined data structures. They point to REST as lacking this capability. According to the philosophical perspective, technically, RESTful web services would be machine-readable because the actions that can be performed are dependent on the media type. Essentially, it comes back to the issue described in the section 'limited representations of state' that bar service providers and consumers from making assumptions about the data returned that is not implicit in the definition of the media type.

From a practical standpoint though, SOAP proponents are absolutely correct. One cannot write their own proprietary media type or data structure and expect it to be machine-readable by clients. If the clients were not specifically programmed with this media type, it would be impossible for machines to both interpret the meaning of arbitrary media types and maintain their claim of using Representation State Transfer (REST).

Structured Data

Similar to the above criticism, in REST, out-of-band knowledge of the response structure is not permitted. Recall the specific structure of the XML user representation from the section on resources. It is challenging

[17] What SOAP calls its URIs

to format and manipulate data if one cannot make assumptions about the structure of a user. When a request for a representation is made, for example, with an `Accept` header of `application/json`, the user is not permitted to assume that the JSON[18] returned will have a list of objects with properties x, y, and z; instead, all that may be assumed is that the response will contain valid JSON.

If a request specifies `application/json`, both of the following are equally valid responses and should be capable of being handled by a client:

```
{
   "id": 3,
   "color": "red"
}
```

and

```
{
   "result": {
      "theId": 3,
      "appearance": "red"
   }
}
```

Both represent the same billiard ball and both are equally valid representations. A consumer of the API cannot assume one will come across the wire over the other, or that any number of other valid JSON objects will not be sent.

Transactions

Another major criticism of REST is that it does not guarantee execution if a call is made. A client may never receive a response to their request. If the client does not receive a response, it is ambiguous whether or not the request was executed on the server. If the request was received and is not idempotent, resending the request might result in undesired consequences. For example, a shopping cart might require a new payment to be `POST`ed to the server. If no response is received, another `POST` might be sent, possibly resulting in two identical charges to a credit card.

[18] JSON is JavaScript Object Notation. A JavaScript Object sent across the web as text.

Many applications require some sort of transaction guarantee. Jumping back to the bank example, funds should only be deposited in one bank account if they can first be withdrawn from another. The reason lost responses are a problem when executing transactions is that there is no method to ensure either a request has been executed or that an error is raised indicating the transaction failed and should be rolled back. The RESTful solution to this is to delegate transaction responsibility to a service. An API consumer would create a new resource representing the transaction.

```
POST /MoneyWireTransaction    →          201 Created
```

Then the transaction can be updated with the transfer amount, source account, and target account. Once complete, the description of the transaction can be submitted to a transaction processor. The transaction processor will execute in a transactional context. Using a transaction processor removes responsibility for detecting an winding back failed transactions from the browser and instead delegates responsibility to the server.

Problems Blockstep Addresses

It would appear REST has already proven itself to be the prominent architecture for the web, if not the optimal for individual applications. The goal of this project is to create a web application that adheres as close to the constraints of REST as is reasonable in creating an online to-do list. Of particular focus are the APIs and how allowing public access to them affects the RESTful nature of the architecture. Three different styles of application will be created, a single-page JavaScript application for a mobile device, a single page browser-based JavaScript application, and a more traditional HTML form page. Some solutions to common criticisms of RESTful APIs will also be explored. Once complete, the quality of the RESTful interface will be evaluated to see if the benefits expected of a RESTful architecture can be realized by the application.

A particular point of focus in the creation of these demo applications will be the definition of data types to pass between the applications and the server's data interface. Many public APIs fall short of being RESTful simply because they assume out-of-band information such as a specific structure of JSON or XML returned by the server. Since JSON is the data format of most public 'RESTful' APIs, applying the proposed draft of JSON schema will be used in attempt to eliminate this assumption by defining a subset of acceptable JSON objects. This may also solve the problem of writing validation rules twice as is required of many applications constrained by a client-server exchange[19].

JSON Schema

JSON schema uses a modified content-type header to indicate the JSON in the payload of the request fits a certain format. The schema defines a subset of acceptable JSON as the content type. While JSON objects are typically passed using the media-type `application/json`, the JSON schema allows a user to extend this to the form of `application/name.of.Class+json`. For example, if a description of a user is defined

[19] For security reasons, user input is always sanitized on the server-side where users cannot subvert the process. For usability reasons, it is considered good practice to validate data before sending it to the server so multiple submissions are not required.

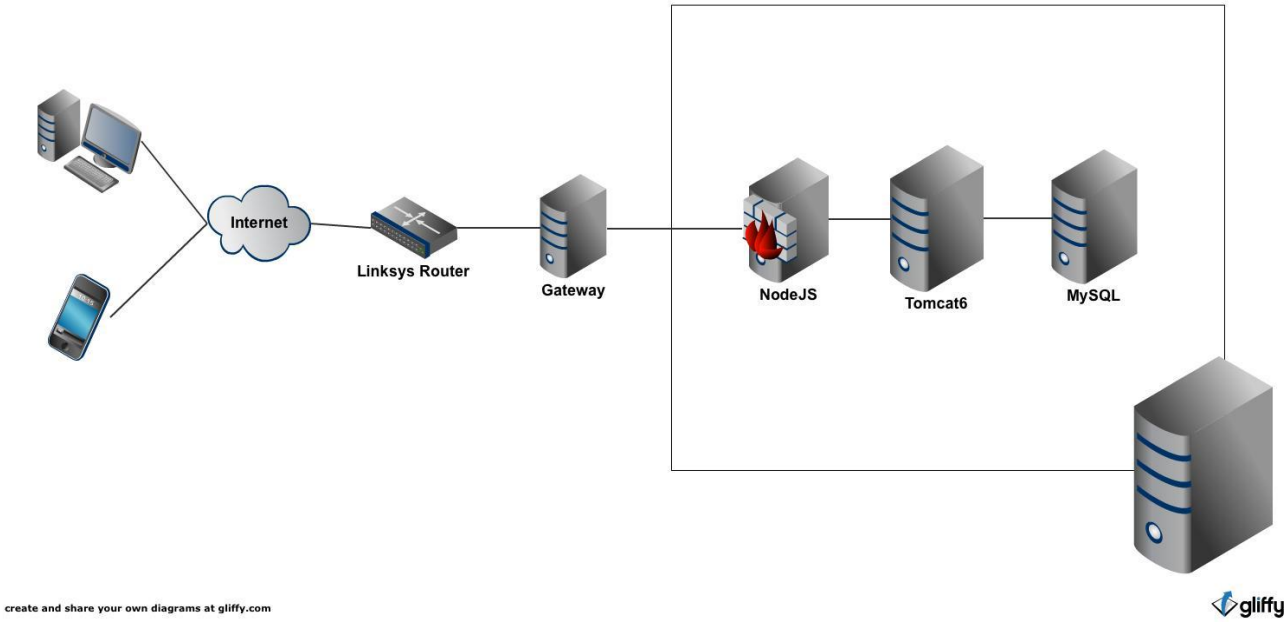
on the domain `example.com`, the media-type would be `application/com.example.user+json`. The schema only specifies how a JSON payload describing a user is formatted.

Unfortunately, the schema suffers from limited adoption and implementation; it is still a draft after all. It has evolved significantly over the past several years without backwards compatibility. The only working validators belong to the two authors of the draft specification, as previously constructed implementations are obsolete due to the evolution of the specification. Additionally, any framework that has incorporated the schema into its code base has not kept up with these changes making them incompatible with other JSON schema definitions. The best example of this code instability is the change from an `optional: false` indicator to a `required: true`. Older validators will not recognize schemas that utilize the latter.

In this project, JSON schema will be used to define the structure of entities exchanged. The schema is not required to read and use the JSON objects as might be the case with an XML schema.

Implementation

Structure overview



The image above depicts the physical domain of the applications. At the far left, the computer with monitor and the phone are the clients. These may be any client on the Internet who navigates to the application's web address. As with any web application, multiple clients may access the site at any given time. The Internet bubble includes all routers, hubs, service providers, etc. who are responsible for relaying data packets between the clients and the router.

Linksys Router

The Linksys router is a piece of hardware that serves as a gateway of sorts to the web application; indeed this application has two. The router has been flashed with Tomato firmware, which allows the device to serve a number of functions.

The first and most important function it serves is to update the IP address in the domain name system. When a user enters a domain name into their address bar, e.g. www.google.cm, facebook.com, anyDomainAddress.org, the browser must look up the IP address of the server hosting the website so it knows whom to contact to retrieve a representation (typically HTML). This look-up is done through the domain name system. Since the IP addresses assigned by Colby to network devices change, the router must update the Internet's address book with the most current IP address of this site.

The second function served is a security function. The router filters traffic on all ports. Ports can be thought of as sub-addresses of an IP address for a particular service. Only ports explicitly opened are reachable from the Internet. For example, the web application should be reachable from the Internet on the standard port 80, but the server hosting the database on port 3306 should only be accessible by computers inside the private network.

For now, the router is configured to forward any traffic on port 80 from the Internet to port 80 of the gateway inside the private network.

The Gateway

The dedicated gateway server[20] is running an instance of Apache HTTPD, the most commonly used server on the web. This instance serves as a reverse proxy by redirecting specific host names to their respective servers. Since the router is only equipped to forward traffic from an external port to a single internal port, a device is needed to separate requests to first.example.com from second.example.com since the sites are hosted on separate machines. Traffic from both sites is sent to the same IP address because both domain entries point to the same IP address.

[20] A server is a computer dedicated to running one or more applications. It typically listens to certain ports for external requests. A web server for example, listens to port 80 for HTTP requests.

A parallel way to think of this is a private mailroom. Many individual people may reside at a street address but when the mail is delivered, a designated person sorts the mail into an in the private mailboxes of the individuals residing within the building based on the name on the envelope. In this case, the street address is the IP address, the designated mail sorter is this gateway server, and the names of the individuals are the host-names of the site (e.g. www.google.com, or my.server.co).

Postal Service	**Internet**
Street Address	IP Address
Apartment Number	Port Number
Name of Apartment Resident	Hostname (`first.example.com`)

The router sends all HTTP mail from the big mailbox on the street to apartment 80's mailbox. The gateway takes all of apartment 80's mail and separates it into mail for Joe and mail for Sam.

The Application Server

In the diagram, one can see three servers all contained in a box. This is to show that, in the project, these three services all run on the same virtual machine - for development convenience. In a larger deployment, it is best to have each operating system running a single virtual machine. Otherwise, if one should fail, it will not drag several other services down with it. Installing each on its own virtual machine would be trivial so it is displayed as such above.

The Firewall

All is not quite as it appears in the diagram. The request, in reality, is sent to the Tomcat6 server which filters requests by forwarding them to the firewall to see if they are valid. If they are not, the filter does not

allow them to pass to the application and instead returns an error to the user. Conceptually, though, the firewall stops malformed JSON schema requests from ever reaching the application server.

The actual firewall, to which Tomcat defers its filtering logic, runs on NodeJS. NodeJS is a web server application that allows users to handle incoming requests and generate responses with JavaScript.

Tomcat6

Tomcat6 is the service that hosts the meat of the application. It is another project of the Apache Foundation and is based in part on the smashing success of HTTPD. Instead of hosting a series of directories, as does the HTTPD project, it allows requests to be handled by Java code responsible for generating responses. The code running on this service makes calls to an external service: the MySQL database.

MySQL Server

The third and final service running on this virtual machine is the MySQL server. This is simply a relational database server for storing and retrieving data records. It was chosen over other SQL servers for no other reason than its ease of use and available peripheral tools [21].

The table below summarizes the overview of services.

Physical Hardware	**Virtual Machine**	**Application / Service**
Router	-	Tomato Firmware

[21] phpmyadmin allows a user to access a web interface for the database and administrate it remotely. MySQL workbench also allows a user to walk through the database schema and modify it if necessary.

Desktop	'reverseproxy'	Apache HTTPD
	'home'	NodeJS
		Tomcat6
		MySQL

The structure of applications, services, virtual machines, and physical devices facilitates the RESTful expansion of these web applications assuming the NodeJS, Tomcat6, and MySQL servers were moved to their own virtual machines. As a requirement however, Tomcat, the service running the application, must be stateless. This will allow duplicated virtual machines hosting the Tomcat service to seamlessly scale the capacity to handle web requests. The only difference would be the distribution of requests to a cluster of Tomcat nodes rather than the one. In a cluster, a reverse proxy distributes requests to multiple nodes; to an outside user, the cluster is perceived to be a single service running on one machine when in reality many nodes split this responsibility.

Similar clustering approaches may be used for the MySQL server to increase capacity. Databases are not as easy to scale or cluster since each member node must modify or read from a shared state. Depending on the needs of an application, different database services will support trade-offs between consistency, query speed, and database integrity. Some databases offer 'eventual consistency' between clustered nodes whereas most structured query databases require strict consistency between nodes to ensure transaction integrity. Eventually consistent databases may have some nodes reporting stale data until they are receive the pertinent asynchronous update. What is given up in consistency is made up for in performance and fault tolerance[22]. Picking the wrong tool for the job is much more palpable when applications are large.

[22] Machines and services will fail eventually. In a distributed system, the application should remain functional when this eventuality occurs. This requires an application to have no single point of failure.

Relational databases like MySQL, Oracle, and SQLite are no longer the only acceptable ways to store data. Map-based, document-based, and various other database structures may be a better fit for the requirements of the project at hand. Which one may depend on an application's read-write ratio, need for consistency, query speed requirements, average query complexity, fault-tolerance, and size of data being stored.

Implementation Details

A much more thorough detailed discussion is warranted for the NodeJS and Tomcat6 servers. It is in these services that the server-side of the RESTful API comes together.

NodeJS Server Implementation

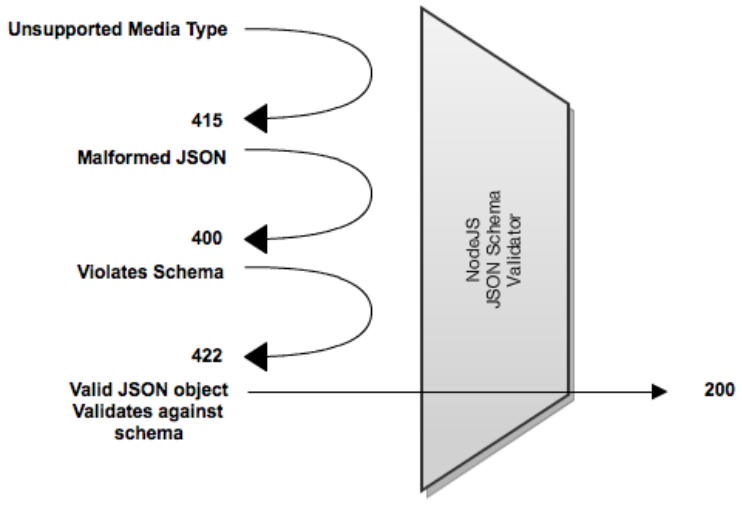
As noted previously, the actual server-side validations of the JSON schema-constrained representations are handled by an instance of NodeJS. As many web servers are structured, any call received by the running service on the specified port will be wrapped in a request object and passed to a 'handle request' method. NodeJS is no exception.

Appendix A contains the code to be run by NodeJS that contains the logic for validating a response. After the required libraries are loaded, the application's schemas are parsed and stored in memory. The server responds to requests like so:

Request	HTTP Response Code	Response Code Meaning
Unsupported media type E.g. `application/some.unsupported.type+json`	415	Unsupported media type
Malformed JSON document	400	Bad request - due to

E.g. a string is missing end quote or an opening bracket omitted		syntax error
Invalid JSON schema object - Request is valid JSON but violates the schema definition (perhaps a required field is missing)	422	Unprocessable Entity
Valid JSON schema object - Request is valid JSON & conforms to the specified schema definition	200	Success

Since the Tomcat6 server forwards these calls, they are not what the original client receives back. Instead, Tomcat returns a response nearly identical to the one it received from the NodeJS server. The only difference is the inclusion of error details. When the filter returns a success status code, there is no guarantee the request returned to the user will also be successful; that depends on the handling of the servlet. If a request is successful, it is permitted to pass through to the actual application.



HTTP response codes will be of little use to the end user of the application but it would be inappropriate for a user to be specifying a request format of `application/.*+json`. These data oriented APIs are designed for asynchronous calls to the server. A developer must write code to generate these Ajax calls. Any reasonable web developer would either validate their requests before sending them to the server, or translate the responses to something meaningful to the end user, usually the former. It is expected that this validator will rarely reject requests. Instead, the purpose of this filter is to prevent malicious users from constructing intentionally malformed requests to exploit the server-side application code. Validation code is provided on demand by the server so that anyone may write their client-side code appropriately.

Web Application Implementation on Tomcat6

The location of the application's business logic, static files, and persistence logic resides in the web archive that is deployed to the Tomcat6 server instance. As previously mentioned Tomcat is a project of the Apache Foundation which allows a developer to programmatically process requests and generate responses. Tomcat serves as a container of the business logic. Requests hit the container and Tomcat bundles them up into neat Java objects and passes them to the appropriate servlet in the container.

The order in which Tomcat makes calls to objects in the container depends on the configuration, as defined by the `web.xml` file, Tomcat's deployment descriptor. Two important types of elements are defined in this file: filters and servlets. When a request is received, it is passed through a chain of filters. As seen in the NodeJS explanation, filters may reject requests entirely. They may also modify or transform requests.

Should a request make it through the chain, it is passed to a servlet. Which servlet the request is passed to is dependent on the mapping of the servlet in the `web.xml` file. For example: a servlet with a regular expression mapping of `/static/*` would handle any request matching that URI. The responsibility of the servlet is to process and respond to the request by modifying the response object provided by the container.

In this project, there are two important components. The filter, described previously, which rejects malformed `application/.*+json` requests based on the NodeJS's validation, is one of the two. The second is a special servlet called the `DispatcherServlet`. This servlet is provided by the SpringMVC framework and forges a connection between the Tomcat container and what will be known as the Spring container.

Spring

The Spring framework is an application platform. It establishes a structure on which other applications may run. The framework is a container of the parts and pieces of the running application. It is responsible for gluing these individual components[23] together, auto-magically. It's not really magic. The container is an inversion of control (IoC) container. In an IoC container, the responsibility for wiring together classes is shifts from a main class the individual component classes. Each component defines one or more public interfaces that may be consumed by other components in the framework. Further, each component defines what it needs of other components. When two or more components are added to the container, Spring 'auto-wires' their dependencies on start-up. Without an inversion of control container, a developer would have to make these connections manually in a 'main' class.

An example of where IoC would be useful:

> A program uses 3 classes `A`, `B`, and `C`.
>
> > `C` requires a reference to both an `A` class and a `B` class.
> > `B` requires a reference to an `A` class.
>
> As it stands now, `A` must be created first, otherwise `B` and `C` will receive null pointers.

```
a = new A();
b = new B(a);
c = new C(a, b);
```

Rather than explicitly instantiate `A` before `B` and `C` in a main class, each class simply defines what it needs. In Spring, this means adding an `@Autowired` annotation to the class's constructor. The container

[23] A component is a module designed to handle a particular function in an application. An example might be a set of code for saving and retrieving objects from long-term storage.

recognizes that classes `A`, `B`, and `C` are required in the application. It also recognizes that `C` requires an implementation of `A` and `B`. The container iteratively creates singletons of the classes as dependencies allow. On the first iteration, an `A` singleton is instantiated, prior to which no other dependencies are satisfied. On the second pass, the container has a singleton of `A` to inject into `B`, so `B` is instantiated as a singleton, and then finally on the third pass `C` is instantiated. Since all of this happens behind the scene, the developer need not worry how to wire all these components together such that all dependencies are met.

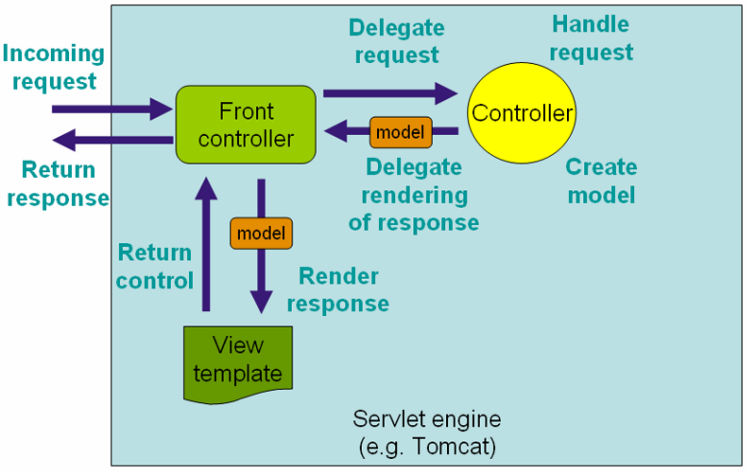
Maven

In the spirit of inverting control by defining dependencies, Maven allows a coding project as a whole to define dependencies on external libraries. Many Java applications, like the one loaded into the Tomcat container in this project, require Java archives that contain classes for use in the application not available in Java's standard library. Some of these archives, like many of the Spring projects, have their own dependencies. Sometimes these dependencies are bundled with the Java archive (jar), other times, they have to be retrieved separately. For the first half of the project, the missing dependencies were located via a stack trace of a `ClassNotFoundException` and googling the target class to see which Java archive was the possessor. Maven, yet another fine project of the Apache Foundation, treats an entire project as an object as described by its configuration file. The `pom.xml`[24] file contains a description of all the dependencies required by the project as well as references to content providers hosting these jars for download. Numerous plug-ins have been written to support development of projects using maven including the ability to build, deploy, and serve a web archive (as Tomcat might do). On compilation, Maven will automatically download the required jar files and pack them into the web archive. This gives developers tight control of their class paths and environment without the headache of hunting down missing dependencies. It also has the perk of being able to send and retrieve the project without all the bulky binaries attached.

[24] pom stands for Project Object Model

SpringMVC

Spring also has a number of sub-projects that can be plugged into the framework. Of particular interest to web applications built atop the framework is SpringMVC. The `DispatcherServlet` facilitates the order of execution in this framework. One might accurately guess that this framework allows the creation of an MVC architecture. Controllers are defined to process requests. The `DispatcherServlet` automatically instantiates these controllers and dispatches requests to the controller method whose mapping matches the request most closely. When the controller method is finished processing, it returns a model and a view name to the `DispatcherServlet` which in turn calls the appropriate view renderer, and finally returns the rendered view to the request's origin.



[25]

Note: The `DispatcherServlet` of the SpringMVC framework implements the functionality of a front controller.

The controller does not act alone however in generating models. Each controller class defines its own dependencies to be auto-wired on instantiation. The most notable of these dependencies are the interfaces provided by the service layer. A number of services allow controllers to create, read, update, and destroy data records. Any application that stores data about or for users must persist this data to long-term storage if it has any hope of achieving Internet-scale. Typically, this is done through a relational database like that of MySQL.

---

[25] Picture from http://static.springsource.org/spring/docs/current/spring-framework-reference/html/mvc.html

Persistence

The database is not a store of Java objects, but rather a series of tabular entries that reference other tabular data. Translating the objects consumed by the application to and from their tabular form is known as persistence. Given the redundant nature of data retrieval, and the number of security holes that might be introduced in the process, it is best not to reinvent the wheel. In a similar vein, transaction management has already been well done by Spring.

In the app marketplace, "there's an app for that"; in the world of Java, "there's a standard for that" (and a framework to match). Hibernate is typically the persistence framework of choice for relational database persistence in Java. It is an implementation of JSR 317, which defines a package of Java interfaces for persistence.

Spring Security

Throughout the entire web application, security is also a concern. A user should only have the right to access and manipulate the data for which he or she has permission. Further, an application dealing with permissions must make sure users are who they claim to be. Spring provides a framework to address these two concerns. Spring-Security is a highly customizable framework for securing URIs, JSP content, or even particular method invocations.

Request Life-cycle Example

To illustrate the life of a request, the following section will outline the process. In this example, a new task is `POST`ed to the `/api/tasks` resource. Consider the following requests:

Request:

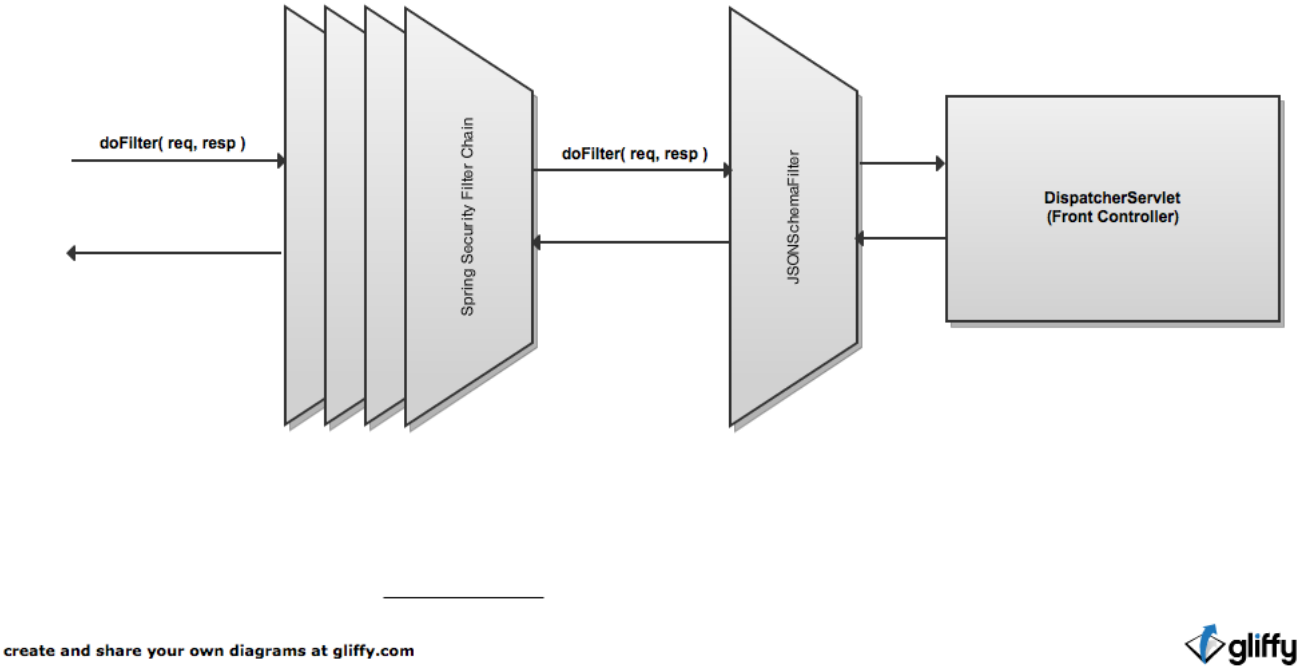
Headers

```
POST /api/tasks/ HTTP/1.1
Host: www.blockstep.com
Connection: keep-alive
Content-Length: 48
Origin: http://www.blockstep.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8)
```

```
  AppleWebKit/535.19 (KHTML, like Gecko) Chrome/18.0.1025.165
  Safari/535.19
Content-Type: application/todo.webapp.dto.TaskDTO+json
Accept: */*
Referer: http://www.blockstep.com/app/
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: JSESSIONID=F842D391673644F07E2D4AF3D4640D60

Body
{"title":"hello world","complete":false,"id":""}
```

Much of this data is not relevant to this discussion but note that all request headers are available to the server and the application it is running. Any of the request headers may affect the processing of this particular request. Ignoring the DNS look-up and packet routing, the request first lands at the port of the Tomcat server. The server reads the request header and creates a new `HTTPServletRequest` object. In addition, it creates a new `HTTPServletResponse` object, which will write to an output stream of the connection. The pair must pass through the server's filter chain before being processed by a servlet.

The container calls `doFilter(request, response)` on the first filter in the filter chain. After the filter has made any necessary modifications to the request it either returns directly or calls `doFilter` on the next filter in the chain. If it should return rather than continue the chain, the request has been rejected. If required, an error code will be translated into a response the client can understand.



The first filter in the chain is actually a separate filter chain in and of itself. Spring-Security provides the Tomcat container with a series of filters that serve a number of functions not limited to session management, 'remember me' authentication, anonymous browser recognition, and a catchall for security errors that may be thrown in the application. Once finished, the last filter calls `doFilter` on the JSON schema filter.

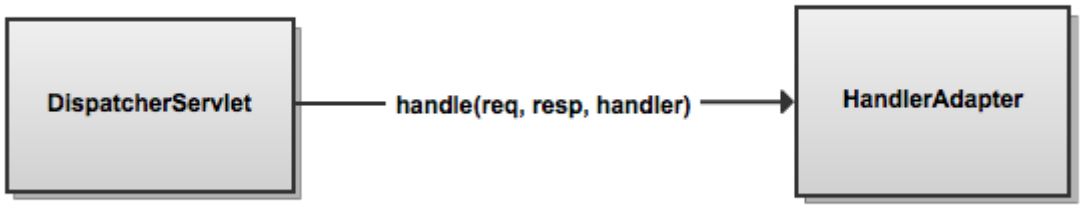
The JSON schema filter checks first to see if the content type of the request being made requires JSON schema validation. If the `Content-Type` conforms to the definition in the JSON schema draft then the request is copied and sent to the NodeJS server for validation. The validation server checks the integrity of the JSON and then proceeds to validate it using the schema specified by the content type. For this particular request, the server has a copy of the `todo.webapp.dto.TaskDTO` definition indicated by the `Content-Type` header. The request payload is valid so the filter will call `doFilter` triggering the container to call `service` on the target servlet.

According to Tomcat's deployment descriptor, `web.xml`, the only servlet that matches the described request is the `DispatcherServlet`. The request and response have now entered the world of SpringMVC. The dispatcher will match the request to the appropriate controller method that will handle the request. The controllers have all been preprocessed at start-up. The controller method this particular request will map to is shown below.

```
@Controller
public class TasksUserController {

    private static final String CLASS_REQUEST_MAPPING = "/api/tasks/";

    @RequestMapping(value = CLASS_REQUEST_MAPPING,
                    method = RequestMethod.POST,
                    consumes = "application/todo.webapp.dto.TaskDTO+json")
    public
    @ResponseBody
    Map<String, Object> post(@RequestBody TaskDTO taskDTO,
              Principal principal,
              HttpServletResponse response) {

        if( ! this.isLong(taskDTO.getId())) {
            taskDTO.setId("");
        }

        Task task = mapper.map(taskDTO,
                               Task.class);



        // Create new Task for User
        User user = userService.getUserByUsername(principal.getName());
        task.setUser(user);
        Long id = taskService.addTask(task);
        task.setId(id);


        // Set response headers
        response.setStatus(HttpServletResponse.SC_CREATED);
        response.setHeader("Location",
                           id.toString());


        // Create response body
        Map<String, Object> jsonResponse = new HashMap<String, Object>();
        jsonResponse.put("success", true);
        jsonResponse.put("id", id);

        ArrayList<TaskDTO> returnArray = new ArrayList<TaskDTO>();
        returnArray.add(mapper.map(task, TaskDTO.class));
        jsonResponse.put("tasks", returnArray);

        return jsonResponse;
    }
}
```

Note that the controller has described what requests should be handled by this method in the `@RequestMapping` annotation. It says, any request to the URI `/api/tasks/` using the HTTP `POST` method that have defined a media-type of `application/todo.webapp.dto.TaskDTO+json` will. This method will not handle `application/json` requests or any other request type. Only requests that specify the JSON schema for `TaskDTO` will be handled by this method. And those requests only make it to processing if they have been validated.

The `DispatcherServlet` calls a method `handle` on an adapter passing it the appropriate handler (the controller method identified above) as well as the request and response objects. The handler adapter then calls the controller method.



Notice the handler method of the controller requires a number of parameters that are not contained in request or response object. The `HandlerAdapter` employs a series of `HandlerInterceptor`s, `MessageConverter`s, and argument resolvers declared in the Spring configuration files. These interceptors will filter the invocations of the handler in much the same way as the Tomcat container's

filters filtered the HTTP requests. In this particular case, the Principal[26] parameter, required by the handler method, will be injected from the Tomcat container. Additionally, the Jackson `HttpMessageConverter` will convert the incoming JSON into a Java object before applying it to the handler's invocation as a parameter.

The handler code first converts the task data transfer object[27] (DTO) to a real `Task`. Since this will be a new task, the current user is retrieved and set as the owner. Then the handler calls on a `TaskService` to add the `Task` to persistent storage. Calling the `TaskService`'s `add` method executes some advice[28] of the `TransactionManager` that opens up a new database transaction. This particular transaction is the database's default because no type was specified. The default is a repeatable read, meaning other transactions can be made concurrently but only concurrently committed inserts will be seen in the context of this transaction, not updates or deletes. Since this particular action is only adding a task to the database, the transaction type makes little difference. The only concern is that it not demand a 'serializable' transaction isolation level as that would disallow all other transactions for the duration of the method's execution. Performance would suffer for little if any gain in terms of data integrity.

Now that the service has initiated a transaction, it calls a method to persist the `Task` on a task data access object (DAO). The DAO uses a dynamic proxy created for this particular application to separate the Hibernate specific API calls from the application's services. The dynamic proxy provides an excellent example of the utility of an inversion of control container. It declares a dependency on an abstract factory

---

[26] The Principal object is set in the Tomcat container by Spring Security and represents the authenticated user. Each request has its own Principal object depending on the user making the request. If a user has not authenticated, an anonymous user principal might hold its place.

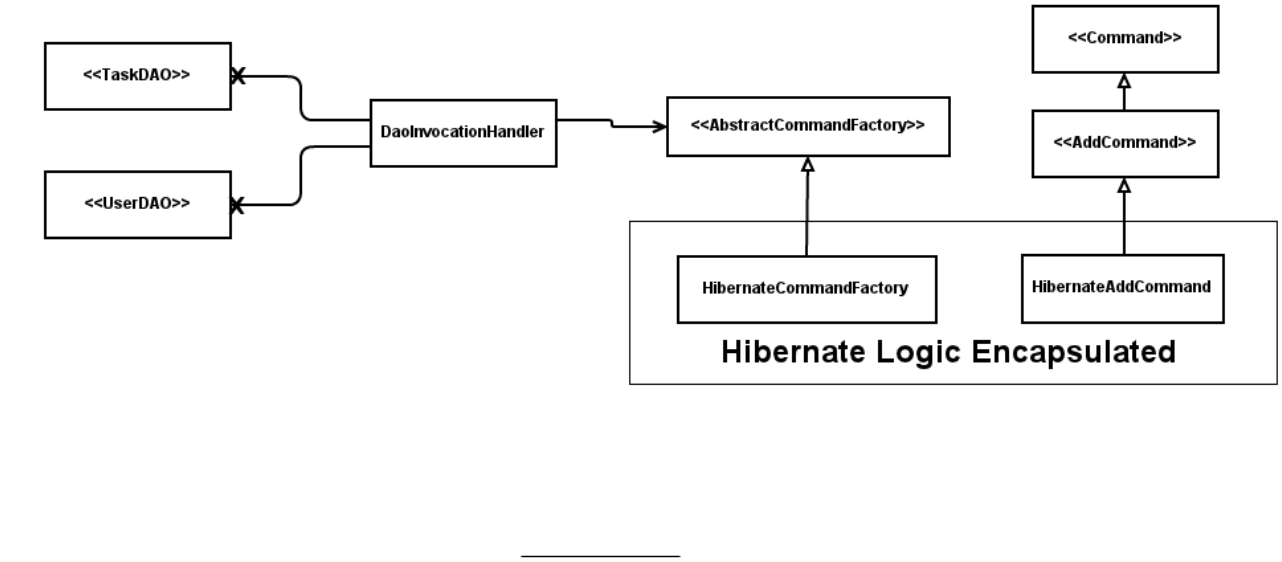
[27] Rather than send and receive a full state of referenced objects, one is only interested in which object is *referenced* so a data transfer object replaces object references with a unique identifier.

[28] Aspect oriented programing allows the description of pointcuts, points in the code where additional code may be executed immediately before and/or after method invocation. Advice is code that is set to run at certain pointcuts.

that is used in creating command objects. Any concrete implementation of the abstract command factory available in the container will be autowired into the proxy. A developer can seamlessly swap out one concrete factory for another without ever changing the wiring of the proxy. Instead, it is simply a matter of including a separate class definition in the container. The factory that is consumed for this project is the `HibernateCommandFactory`. It generates commands that encapsulate Hibernate persistence logic.

The reason for separating data access logic from the persistence implementation is because Hibernate does not strictly follow JSR 317, Java's standard for persistence. Instead it provides a number of additional 'nice to have' methods. It may be the case that another persistence library is used in the future, as Hibernate is not known to structure its SQL queries efficiently. In order to manage this risk, all Hibernate specific API calls are wrapped in `Command` objects to be executed by this dynamic proxy. The proxy knows nothing of how the persistence is performed. In order to swap persistence mechanisms, the existing command factory must be replaced with a command factory for the desired persistence architecture.

This package utilizes a number of design patterns only mentioned in passing thus far. A command pattern is used to execute individual additions, insertions, updates, or deletions on the database. A factory implements an abstract factory for the instantiation of command objects to be called by the dynamic proxy. A dynamic proxy is responsible for backing numerous DAO interfaces and executing the commands appropriately. All the Hibernate logic and API calls are contained in the command objects.



In this case, the dynamic proxy will recognize a call to the `TaskDAO` to add a task. It will tell the wired `AddCommand` implementation to execute itself with the object passed. The concrete class of this command is the `HibernateAddCommand`, which will call Hibernate's, `save` method on the task. Hibernate will use the mappings defined on the `Task` class definition to persist the object by generating the required SQL and executing it on the plugged in MySQL database. The method returns an identifier for the newly created `Task`. This will return all the way up through to the `TaskService`. When the handler, `TaskUserController`, receives this ID, it will set it in the `Location` header of the `HttpServletResponse`. The status code will then be updated to `201 CREATED`. This will indicate to consumers of the API that a new instance has been created at the location specified by the `Location` header. A map is constructed detailing the operation. This map is to be returned to the initiator of the request as the body of the response. The controller returns the `HashMap`, which is intercepted by the Jackson `HttpMessageConverter` and written out to the `HTTPServletResponse` as JSON. Now that the handler has finished executing, the `DispatcherServlet` will recognize that the response has already been written by the message converter and will not try to render an additional view. The request has been completed.

The intent of walking a request through its life-cycle is to illustrate the layered nature of the application. Fielding calls for a layered service in order to promote loose coupling in the application. In many respects this is focused on the layered nature of the distributed system. There same loose coupling can be achieved at an application level as well with a layered approach. Many web applications, particularly those implemented in Java utilize this separation of concerns.

Another intention of this section is to show that each layer and the application as a whole maintains almost no data across requests. Each request is handled with only its headers and body as the context for processing. And this is almost entirely true. The only component that keeps this constraint from being fully realized is the Tomcat6 container and Spring-Security. Spring Security relies on cookie data and a session id provided by Tomcat to 'remember' who it is that is making the request across the session. Eliminating this dependence is an opportunity for future work. Cookies are rarely used in a RESTful manor and for that reason should be avoided. However, skrebbel describes an elegant way in which this might be done on StackOverflow.[29]

The Client-Side

The client-side of this application has three main application fragments. The traditional HTML page application type accounts for the main page, registration, and login/logout facilities. Once logged in, the user will be directed to either the mobile application or the browser application depending on their user device. All three of the following sub-sections exchange data with the server's JSON schema API.

Registration

The registration page of the site is the most interesting because the data is validated client-side using the served form definition. A JSON schema `application/todo.webapp.dto.RegistrationForm+json`

[29] http://stackoverflow.com/questions/319530/restful-authentication

defines the required form of the registration form. While the user does not see this document, the error messages generated are used directly on the page to indicate how the user should revise the form. This form is implemented using the Dojo framework.

JSON schema

As was previously discussed, JSON schema is a way of specifying a subset of acceptable JSON - the JSON that conforms to the given schema.

The Dojo Framework

Initially, the Dojo framework was chosen as the client-side JavaScript framework because it included both a JSON schema validator and a number of facilities for manipulating and displaying JSON-formatted data to users. The curators of the framework are the employees of SitePen Inc., which, among them includes an authors of the JSON schema draft specification. Naturally, the schema-based validation was incorporated into the framework. Unfortunately though, Dojo's validator has not kept pace with the schema draft specification and is incompatible with the version capable of running on NodeJS[30].

For this project, an extension to the Dojo framework was composed in order to expedite the creation of JSON schema-backed forms and remedy the inconsistencies between available validators. This extension, found in Appendix B, loads the specific JSON schema that backs the form as well as a statically served a copy of Kris Zyp's[31] most recent validator (as opposed to the one provided by Dojo). It then redefined the form's submit event to first validate the form. If any errors are found during validation, it displays them and arrests the submission event. Only after the form validates does it create an Ajax request to the server. Note that jQuery's highly customizable Ajax method is used to create and send the request to the server.

---

[30] Dojo requires some document and window environmental variables to be initialized in order to function. Since these are not available on NodeJS, an alternative validator was required.

[31] Kris Zyp is one of two authors of the JSON schema draft specification and has made his validator publicly available on github

This worked well for a single JSON schema constrained interaction. However, when maintaining a list of tasks constrained by JSON schema using Dojo-provided data structures, it was not possible to unobtrusively modify the Ajax bindings used to exchange data with the server. Specifically, it was not possible to change the content-type request header from `application/json` to `application/some.schema+json`. A different JavaScript framework was required or the data structure would need to be reinvented.

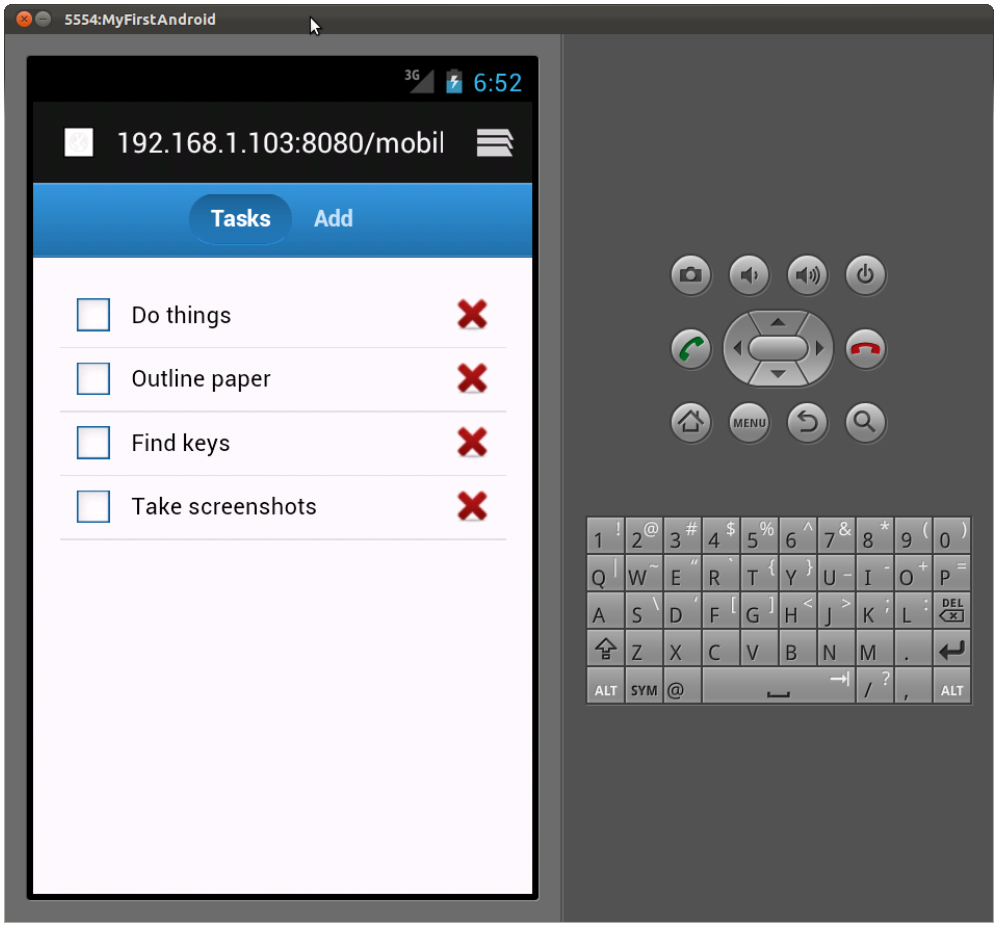
Point of Extension

Originally, the registration process was designed to be a multi-page form. Clicking the 'create account' button would make a `POST` to the registration resource to create a new form instance on the server. Then as the user completed a page, a `PUT` request was made to the previously create form instance. Each time it would validate the form; if data was missing or erroneous, it would send the user to the appropriate page to enter or correct data. Finally, when the registration form validated completely, a `POST` was made to the `/api/users` resource with a reference to the completed registration instance. This would have demonstrated that a developer can achieve session-like functionality by appropriately forming their resources.

This was not realized in this application though because JSON schema does not mesh well with partial objects. Some fields would be required but not filled in on the first page. The separately validating NodeJS server would have rejected a submission of this resource. Several JSON schemas would have been required, one for each page. And additional calls would have been required in the servlet's processing to determine where in the process of validation a resource stands.

Mobile

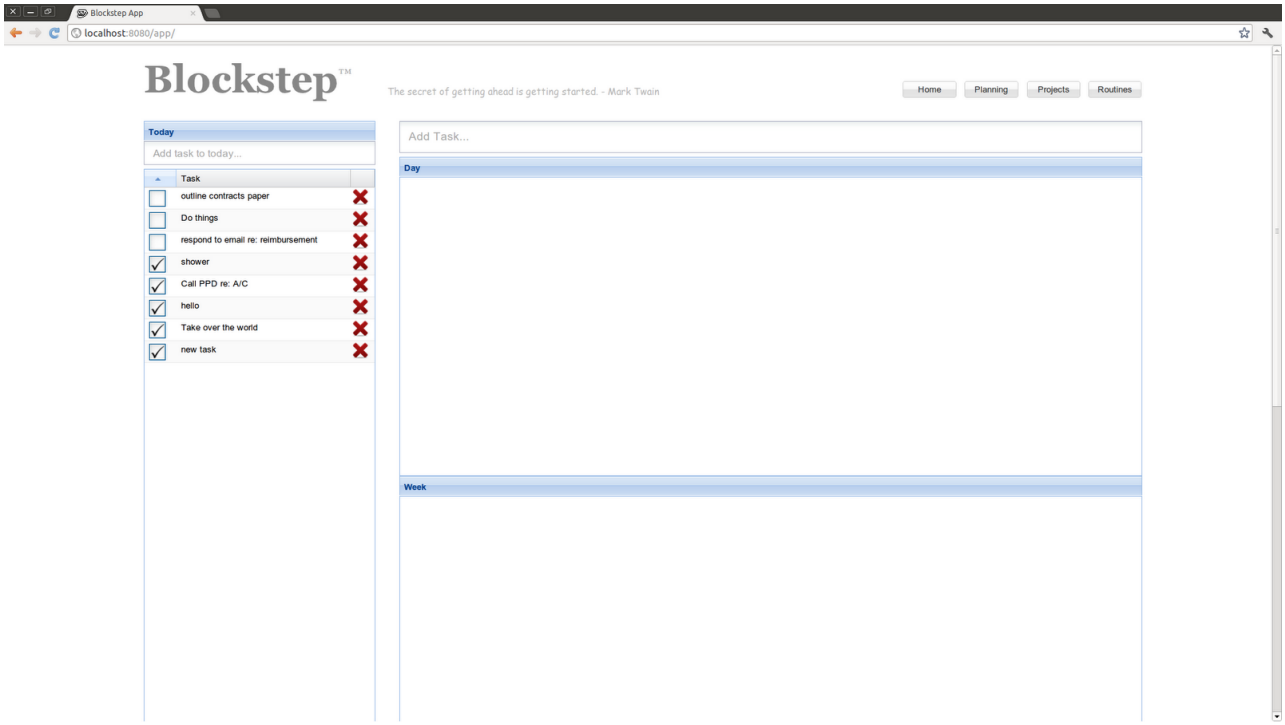
The mobile application is a very simple application that allows a user to add tasks to a list, and then check them off or remove them. It persists these tasks and changes to the server using the REST-styled resources previously constructed on the server-side.



Since the issue of transporting representations with Dojo could not be remedied by extending the classes, Sencha's Touch platform was attempted. It did not contain a way to modify the request header to change the content type of `POST` and `PUT` requests as required but it's much more wholesome architecture not only allowed, but even encouraged extension and customization.

Browser

The browser essentially has the same behavior as the mobile phone. It is implemented in ExtJS[32] requiring a second method of formatting requests between the data-structures and the server-side API. The intent of the browser app is to demonstrate that multiple applications can consume the same server-side resource-based interface without much modification.



[32] Also produced by Sencha, ExtJS contains many of the same classes and structure as that of Sencha Touch.

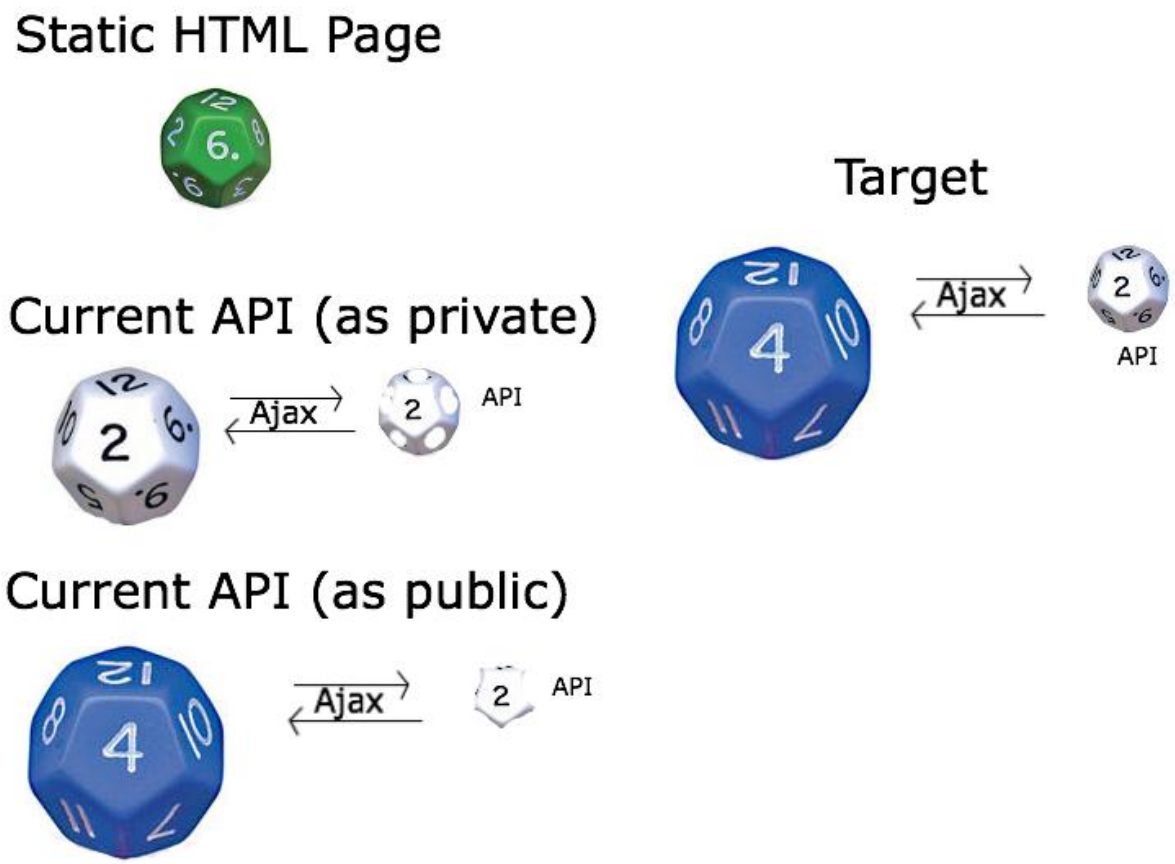
Evaluation

Whether or not this application was successful in attaining an architecture that conforms to the specifications of representational state transfer falls on the determination of whether or not the API is publicly available or not. To clarify, the API is composed of the CRUD operations for `Task`s and `User`s under the `/api/*` URI. If this API is not publicly available, the calls made from the application are executed by the application's own code-on-demand. Since the developer of the site wrote this code, he may make assumptions about the structure of requests and responses as well as utilize the API to modify state. The public never directly affects state through calls to the application's resources but rather by interacting with the representation presented to them in conjunction with the code on demand. This particular application was a compilation of CSS, HTML, images, and JavaScript. The key point is the user navigated state transitions via interaction with the representation presented, not the API. If the API is public, calls to the CRUD operations are no longer from the API author's code-on-demand. Consumers of the API who make their own code-on-demand to utilize the API make assumptions about the state transitions available. In order to correct this, hyperlinks would need to be included in the representations received from the server.

A simpler way of thinking about this is to picture a 12-sided die.

The representation presented, the 12 in this case, is the current state of the application. There are 5 possible transitions to neighboring sides that can be made from this particular representation as can be seen on the peripherals of the die. The web-based parallel would be an HTML page displaying a black 12 with 5 links on the page. Clicking any of these links will change the application's state; clicking the 7 for

example would direct the browser to a new page with a 7 displayed with 5 new links. A similar model must be implemented as an API.



The figure above depicts the current state of this particular application. For comparison's sake, the representation of a static HTML page is included. A user makes changes to the static HTML page by clicking any of the five links, the equivalent to rotating the die to any of the 5 visible adjacent sides. As a private API, the larger die on the left represents the page displayed to the user. This page includes code-on-demand owned by the application hosting the API. As the user interacts with the current rendering of the site's state, the code-on-demand is turning the smaller die to the right in order to change the state of

the application. Since both the API and code-on-demand are developed in parallel, the available state transitions are known even though they are not declared by the Task and User representations. If an outside application wished to change the state of the API / rotate the die, as shown by the blue client-side die, they would be flying blind because they cannot make the same assumptions about available state transitions. It is as if their code cannot see the adjacent sides of the smaller white die representing this API. To remedy this, each representation must indicate what transitions are available in the data type so that external code on demand is capable of turning the die/changing the state.

As far as the success of JSON schema, the malleable content-type it provides is a step in the right direction. In the above example, the JSON schema allows the consumer of the public API to know, at least, what the current representation looks like. E.g. it is known that the current state displays a number 2 and has a white background. Without JSON schema, public code-on-demand would be making calls to an invisible die.

Summary

In the construction of private web applications, developers should keep in mind the principles of REST. An application's architecture may not adhere to all constraints of REST, but that does not mean it is a bad architecture for the task at hand. The catch is that the application cannot be labeled as REST until it fulfills all the criteria. The only way to keep an application RESTful is to ensure it is hypertext-driven. The API calls might not be hypertext driven, but these can be wrapped by code-on-demand.

As each constraint required by REST is supposed to shape the code to elicit certain qualities, using some, if not all of the constraints, may provide some of the attributes sought by REST. The closer to REST an application's architecture is the better fit for the web it likely will prove.

Blockstep demonstrates that JSON schema is a viable solution for validating assumptions about payload structure. In order to fully realize a RESTful architecture, future versions of Blockstep must be hypertext-driven. Blockstep's service API provides evidence that a uniform interface can be plugged into a number of consumers without modification. Sencha Touch, ExtJS, and Dojo[33] were all able to connect to Blockstep's API without modification, and manipulate the application's state. [34]

[33] Recall that these 3 libraries are client-side JavaScript libraries that consume RESTful web services

[34] Sencha Touch and ExtJS were modified later to accommodate the new data-format JSON schema only to change the `Content-Type` headers submitted.

Glossary

Ajax - Asynchronous client-server interactions. Often implemented over HTTP using JavaScript.

Apache HTTPD - The most successful web server. Based on file directory structure.

API - Application Programming Interface - an interface made available to other applications in order to manipulate the state of application offering the interface.

Application - A program designed to assist a user in accomplishing a task. Network applications include code running on all participating nodes, when a client access a website, their browser becomes an extension of the application.

Application server - The server of application representations. This part of the program is responsible for generating responses to requests.

Architecture - A set of constraints placed on a design to elicit certain properties. Applies as much to building buildings as it does to software construction.

Asynchronous - Not synchronous. E.g. a client browser does not wait for a response from the server to continue working. Upon receipt of a response, the browser will be notified.

Body - The payload of an HTTP request.

Browser - A client application for traversing the web. The application is responsible for generating requests and rendering responses.

Cache - A temporary and transparent storage that allows saved response values to be returned rather than waiting for a new response to be generated.

Client - The application that creates calls to a server.

Cluster - Several nodes that operate in collaboration with one another to provide a single service.

Constraint - A limitation placed on a design to emphasize certain qualities.

Content-Type - An HTTP header indicating the format of the payload in the body.

Controller - One element in an MVC architecture that is responsible for handling the business logic of the application.

Cookie - A text fragment assigned by a server to a browser for use in subsequent requests.

CRUD - Acronym for create, read, update, destroy.

CSS - Cascading Style Sheets - Markup defining how HTML elements should be rendered.

Dependency - Declared reliance on an external interface.

Domain name - A string that identifies a website. (e.g. www.google.com)

Filter - An object responsible for modifying or rejecting requests to a service.

Firewall - Hardware or software responsible for controlling the flow of data into and out of the network.

Framework - A platform on which applications can be built. Usually, frameworks provide a number of supporting APIs.

Gateway - A point of entry or exit, particularly in relation to proxying requests.

Header - A key value pair of strings attached to the top of an HTTP request or response.

HTML - Hypertext Markup Language, the backbone of browser views.

HTTP - Hypertext Transfer Protocol, the application protocol used by the web.

HTTPD - *see Apache HTTPD*

Hypermedia - A super-set of hypertext that also includes multimedia (audio, video, etc.)

Hypertext - Textual documents that contain references to other text-based documents.

Idempotent - Application of an idempotent object does not affect the object after the first application.

Interface - Set of hooks into an application made available.

Intermediary nodes - Nodes between a client and server that are responsible for relaying requests and responses.

IP address - The Internet protocol address of a network device. Either 4 or 6 bytes long. (Typically written in the form 137.146.138.123)

JSON - JavaScript Object Notation. A message format that parses directly into a usable JavaScript object.

Media-type - The format of the a stream of bits.

Model - In an MVC architecture, the model represents the state of an application.

Node - A network device, virtual or real.

Origin server - the server from which the content of a website originated.

Out-of-band information - Information assumed by a developer that cannot be inferred from the uniform interface.

Payload - The series of bits following the headers of an HTTP request or response.

Persistence - The process of saving and retrieving objects to and from long-term storage. In this project, converting Java objects to a tabular form and back again.

Port - A sub-address on a network device that an application might be listening on. E.g. HTTP servers listen to port 80 for HTTP requests; they will not receive requests to port 81.

Representation - A snapshot of state in a particular format.

Request - A message generated by a client to trigger a reaction (response) from a server.

Resource - A URI that describes a noun.

Response - A message generated by a server to respond to a client's request.

Safe - A safe method does not affect state, it is read only.

Schema - A formal description of a data format.

Server - A machine hosting a service that listens to a port for requests.

Servlet - A Java object responsible for handling and responding to `ServletRequest`s with `ServletResponse`s.

Session - Data shared across multiple requests from the same client application.

Set - A collection of unique values.

Standard - A specification to which many development organizations agree to adhere to in the interest of promoting reusability and interoperability.

State - A particular configuration of data, particularly in relation to an object.

Stateless - No data is retained by an object between requests.

Subset - A fraction of a set.

Transaction - An all or nothing, indivisible, atomic unit of work. Everything in the transaction must complete, otherwise nothing completes.

URI - A location on the web of a particular resource

URL - A location on the web of a particular resource that also defines the representation.

E.g. http://www.example.com/book.html is a URL because it specifies an html representation

http://www.example.com/book is a URI, not a URL because no representation is specified

View - A view is the format and rendering of data to a user. This takes place in the browser for web-based applications.

Virtual Machine - A virtual machine is a subsection of a physical machine that shares physical resources with other co-located machines. Many virtual machines may exist on one physical box but their time on the processor, RAM, and hard-drive space are partitioned. Each virtual machine must run its own operating system.

XML - Extensible markup language, a markup language for defining other markup languages.

References

Asaravala, Amit. "Giving SOAP a REST." DevX. QuinStreet Inc., 21 Oct. 2002. Web. 05 Apr. 2012. <http://www.devx.com/DevX/Article/8155>.

Fielding, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. Thesis. University of California, Irvine, 2000. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 4 Jan. 2001. Web. <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf>.

Fisher, Paul T., and Brian D. Murphy. *Spring Persistence with Hibernate*. [New York]: Apress, 2010. Print.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. Print.

Johnson, Rod, Juergen Hoeller, and Keith Donald. "Reference Documentation."*Reference Documentation*. Spring Source, VMware. <http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/>.

"RFC 2046 - Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types."*RFC 2046 - Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. Network Working Group, The Internet Engineering Task Force, Nov. 1996. Web. <http://tools.ietf.org/html/rfc2046>.

"RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1." The Internet Engineering Task Force, June 1999. Web. <http://www.ietf.org/rfc/rfc2616.txt>.

"RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax." *RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax*. Network Working Group, The Internet Engineering Task Force, Jan. 2005. Web. <http://tools.ietf.org/html/rfc3986>.

Richardson, Leonard, and Sam Ruby. *RESTful Web Services*. Farnham: O'Reilly, 2007.*Safari Books Online*. Web.

Zip, Kris, SitePen (USA), and Gary Court. "A JSON Media Type for Describing the Structure and Meaning of JSON." *Draft-zyp-JSON schema-03 - A JSON Media Type for Describing the Structure and Meaning of JSON Documents*. Internet Engineering Task Force, 22 Nov. 2010. Web. <http://tools.ietf.org/html/draft-zyp-JSON schema-03>.

Index

A

Ajax, 8
API, 17
architectural constraints. *See* architectural style
architectural style, 12
autowire, 40, 49

C

Cache, 13, 21
Cascading Style Sheets, 6
client-server, 24
cluster, 36
`Content-Type`, 7
Cookies, 51
CSS. *See* Cascading Style Sheets

D

document type, 4
Dojo, 52
dynamic proxy, 48

E

ECMAScript. *See* JavaScript
eventual consistency, 36

F

Fielding, 12, 15, 23, 26, 51, 64
filter, 44
Firewall, 34
firmware, 32
framework, 43

H

Hostname, 34
HTML5, 4
hypertext documents, 10

I

idempotent, 19, 21, 28, 61
Intermediary node, 21
Internet Engineering Task Force. *See* IETF
inversion of control, 40
IP addresse, 33

J

JavaScript, 8, 22, 26
JSON schema, 30, 35

L

layer, 14
limited representations of state, 14, 23

M

MySQL Server, 35

O

out-of-band information, 23, 24, 27, 30

P

payload. *See* response body, request body
persistence, 42
port, 33
Principal, 48

R

representation, 16
Representational State Transfer. *See* REST
resource, 15, 22
REST, 10
RESTful interface. *See* Uniform Interface
Router, 32

S

safe, 19
server, 5
service descriptor, 11
SOAP, 10
source, 4
standard, 43
static markup, 8

T

Tomcat6, 35
transaction, 11, 28
transport protocol, 12

U

uniform interface, 19, 21, 22, 26
uniform resource identifier. *See* URI

V

virtual machines, 36

W

web service description language. *See* WSDL
web services, 12

Appendix A - JSON Schema Validator on NodeJS

```
// Require package
var fs = require('fs');
var validate = require('../static/js/libs/json/schema/lib/validate.js').validate;
var http = require('http');

// Dictionary of JSON schemas
var schemaDictionary = {};

// Path to schemas / directories
var realSchemaDirPath = fs.realpathSync("../static/json/schema/");
var schemaDirPath = "../static/json/schema/";
var schemaDir = fs.readdirSync(schemaDirPath);

// Populate schemaDictionary with schemas
for (var file in schemaDir) {
    var contents = fs.readFileSync(realSchemaDirPath + '/' + schemaDir[file],
                                   "UTF-8");
    var key = schemaDir[file].toLowerCase();
    console.log(key);
    schemaDictionary[key] = eval('(' + contents + ')');

}

function getTypeFromContentType(contentType) {

    console.log(contentType);
    var entityTypeRegex = /application\/(.+)\+json.*/;

    var match = entityTypeRegex.exec(contentType);
    return match[1].toLowerCase();
}

http.createServer(
    function (req, res) {

        var content = "";

        req.addListener("data", function (chunk) {
            content += chunk.toString();
        });

        req.addListener("end", function () {
            var json;

            // Attempt parse, reject if invalid json
            try {
                json = JSON.parse(content);
            } catch (err) {
                res.writeHead(400, {});
                res.end();
            }
```

```
            // If no entity is passed - return 400
            if (json === undefined) {
                res.writeHead(400, {});
                res.end();
                return;
            }


            // Get appropriate schema
            var schemaName = getTypeFromContentType(req.headers["content-type"]);



            var schema = schemaDictionary[schemaName];
            console.log(schemaName);
            console.log(schema);

            // If schema is not defined, return unsupported
            if (schema === undefined) {
                console.log("Schema undefined!");
                res.writeHead(415, {'Content-type':'application/json'});
                res.end();
                return;
            }

            var report = validate(json, schema);

            // If entity is valid, send success
            if (report.errors.length === 0) {
                console.log("Valid");
                res.writeHead(200, {'Content-type':'application/json'});
                res.end();
            } else {
                console.log("Invalid");
                // Else, send errors
                res.writeHead(422, {'Content-type':'application/json'});
                res.end(JSON.stringify(report.errors), "UTF-8");
            }


        });
    }).listen(3333, "127.0.0.1");
console.log('Server running at http://127.0.0.1:3333');
```

Appendix B - Dojo Form Extension

```
dojo.provide('todo.Form');

require(['dojo/_base/declare', 'dojo/dom-form', 'json/schema/lib/validate'], function (declare)
{
    declare("todo.Form", null, {
        constructor:function (/*Object*/ args) {

            this._id = args['formId'];
            this.formHandle = $("form#" + this._id);

            // Return a new form object
            this.method = "POST";

            var location = this.formHandle.attr("JSON schema");
            this.getSchema(location);

            this._validators = {};
            this._responseHandlers = {};
            this.contentType = "application/json";

            var handler = {
                todoForm:this,

                handle:function handle(event) {
                    dojo.stopEvent(event);
                    if (this.todoForm._validate()) {
                        this.todoForm._send();
                    }
                }
            };


            dojo.connect(dojo.byId(this._id), "onsubmit", function (event) {
                handler.handle(event);
            });
        },

        setContentType:function (type) {
            this.contentType = type;
        },

        setCustomValidation:function (validators) {
            this._validators = validators;
        },

        setMethod:function (method) {
            this.method = method;
        },

        setResponseHandler:function (handlers) {
            this._responseHandlers = handlers;
        },
```

```
        _validate:function () {
            this._clearErrors();

            var isValid = true;

            isValid = isValid && this._jsonSchemaValidation();

            isValid = isValid && this._customValidation();

            return isValid;
        },

        _customValidation:function () {
            var isValid = true;
            var instance = dojo.formToJson(dojo.byId(this._id));

            for (v in this._validators) {
                if (!this._validators[v].test(instance)) {
                    this._setError(this._validators[v].error);
                    isValid = false;
                }
            }

            return isValid;
        },

        _jsonSchemaValidation:function () {

            var instance = dojo.formToJson(dojo.byId(this._id));
            var errors =
                json.schema.lib.validate(JSON.parse(instance),
                                         this._schema).errors;

            for (error in errors) {
                this._setError(errors[error]);
            }

            if (errors.length === 0) {
                return true;
            }
        },

        _setError:function (error) {
            var errorDiv = $("#" + error.property).siblings("div.errors");

            if(errorDiv.length === 0) {
                console.log("Undisplayed Error - Property: " + error.property + " Message: " +
error.message);
            }

            errorDiv.text(error.message);
        },

        _clearErrors:function () {
```

```
            $(".errors").text("");
        },

        getURI:function () {
            return this.formHandle.attr("action");
        },

        _send:function () {
            $.ajax(this.getURI(), {
                data:dojo.formToJson(dojo.byId(this._id)),
                contentType:this.contentType,
                type:this.method,
                statusCode:this._responseHandlers
            });
        },

        getSchema:function (location) {

            var handler = {
                form:this,
                handle:function (data) {
                    this.form._schema = data;
                }
            };

            $.ajax(location, {
                dataType:"json",
                success:function (data) {
                    handler.handle(data)
                }
            });
        }

    });

});
```