TiddlyWeb: HTTP for Tiddlers

Chris Dent

Peermore Limited

Newbury, United Kingdom

cdent@peermore.com

ABSTRACT

TiddlyWeb was created as a web-based storage system for TiddlyWiki, a single-user all-in-one-HTML-file wiki. TiddlyWeb answers the question: Using what we know about the open web, and especially HTTP, how would a system that supports multiple users and sharing of TiddlyWiki content be designed and built? The answer leads to a resource oriented design. TiddlyWeb builds on learning from previous systems. Its design and implementation illuminates useful patterns in service development and highlights common wisdom in systems design. Though sometimes described as a RESTful store for tiddlers, it is perhaps more appropriate to call it an HTTP store. Rather than going into detail on the architecture of TiddlyWeb, this paper reflects on the lessons learned during development.

Keywords

REST, HTTP.

1. INTRODUCTION

The Representational State Transfer (REST) architectural style was derived from, was established in concert with**,** and has driven the advancement of the World Wide Web and HTTP [2]. As such, to ask if a service presented over HTTP is RESTful or not is to ask the wrong question; a more useful question would be to ask to what extent a service avails itself of the opportunities and advantages provided by the principles and constraints of the architectural style. TiddlyWeb[1] is a service designed from the outset to follow REST, while at the same time pragmatically bending the rules where real-world implementations and situations present conflicting constraints. How these constraints impacted the design and development of TiddlyWeb confirm REST principles and provide anecdotal evidence of useful patterns for the design and development of other systems.

[1] http://tiddlyweb.com/ Also see the Appendix.

2. TIDDLYWEB

TiddlyWeb builds on the shoulders of its predecessors. Jeremy Ruston created TiddlyWiki in September of 2004 [6] to automatically wikify paragraphs of content in an HTML page. Over time it "folded in on itself to become a completely self-contained wiki engine" [Ruston, personal communication]. TiddlyWiki makes extensive use of JavaScript and DOM manipulation to create a wiki in a single HTML file. Individual chunks of content are called tiddlers. Links are made between tiddlers in the same fashion as between pages in a standard wiki. Jeremy released TiddlyWiki as an open source project and a thriving community has been improving and extending it with plugins ever since. Many users have wanted to keep their TiddlyWiki files on network-based servers. Tools called server-sides have been created which allow parts or all of a TiddlyWiki file to be posted to a server location. Of these, the most visible have been ccTiddly [2] and Tiddlyspot[3].

In the summer of 2006, developers at Socialtext started work on what was called a REST API [7] for the Socialtext wiki product. The goal was to provide a web service that allowed external systems to get data in and out of Socialtext. Later that summer, Socialtext started working with Jeremy Ruston and his company, Osmosoft, to create a service called Socialtext Unplugged which would pair TiddlyWiki with the Socialtext API to allow offline reading, editing, and synchronization of Socialtext workspaces. Each system helped to influence the other. Code in TiddlyWiki that could be used to synchronize with remote services reached a level of maturity that now allows it to interact with several different wiki systems, and other networked services.

Conversations ensued, over many months, exploring the idea of creating an application, with a REST API following the Socialtext model, that would explicitly support something that was missing from other TiddlyWiki server-sides: effective reuse of existing tiddlers, including other people's tiddlers, to compose new information resources. This implied two important changes: 1) Prioritizing individual tiddlers as first class resources in the artifact world, and 2) Creating resources that allow for the grouping and composition of tiddlers into useful collections.

2.1 Reuse

Reuse is the overarching design goal of TiddlyWeb. Meeting that goal drives all decisions about how to assemble functionality and leads to a basic principle: make TiddlyWeb be like the web.

[2] http://tiddlywiki.org/wiki/CcTiddly

[3] http://tiddlyspot.com/

If tiddlers[4] are to be reused, they need to be on the web, addressable by URLs. If tiddlers are to be reused by people they need to have good names that humans like. If tiddlers are to be shared between people there must be a mechanism to disambiguate names and to manage access. This leads to TiddlyWeb bags[5]. If tiddlers are to be composed from multiple sources there must be a mechanism to compose and select desired parts. This leads to TiddyWeb recipes[6] and filters[7]. If tiddlers are to be reused in different contexts and different representations then some form of content-negotiation must be provided by the service. If the service is sending and accepting different representations of resources, then it should have a system for serializing and de-serializing those representations from and to a persisted form. If the service is to be flexible in the face of unexpected uses, then the serialization system should be extensible. If the serialization system is extensible then developers will want to work in many contexts to create their extensions. If there are many development contexts then the service needs to support multiple ways of being hosted on a network and multiple ways of persisting resources to storage.

Resolving these goals points a clear path to a design that is resource oriented, scalable and extensible, and which demonstrates the effectiveness of exercising HTTP. The design points out some areas in which a pragmatic rather than dogmatic approach wins the day. Many of the lessons are not new but a recapitulation or repackaging of well known ideas in systems design.

2.2 Lessons

TiddlyWeb's original description is "[a]n optionally headless, extensible RESTful datastore for TiddlyWiki"[8]. Comparing the design, development and use of TiddlyWeb with the characteristics and constraints of the REST architectural style shows the extent to which those constraints have been met, been useful, or been rejected in favor of pragmatic concerns. Knowing specific areas where compromises or adaptations have been made may be useful for developers of other systems that wish to take advantage of the architectural style.

2.2.1 REST

REST is derived via six architectural styles (seven if including the null style) that form the constraints of a REST system [2].

[4] In TiddlyWiki and TiddlyWeb, a tiddler is the name for a single addressable piece of content. It might represent a single idea, a topic summary, or a blog posting. Importantly it can also be a chunk of Javascript code that is used to extend the functionality of the TiddlyWiki in which it resides. TiddlyWeb adds to this by allowing tiddlers to store any content that can be assigned a MIME type. The content then becomes accessible over the web with a tiddler URI.

[5] http://tiddlyweb.peermore.com/wiki/recipes/docs/tiddlers/bag

[6] http://tiddlyweb.peermore.com/wiki/recipes/docs/tiddlers/recipe

[7] http://tiddlyweb.peermore.com/wiki/recipes/docs/tiddlers/filter

[8] http://pypi.python.org/pypi/tiddlyweb

2.2.1.1 Client-server

The canonical UI for TiddlyWeb is in TiddlyWiki. TiddlyWeb is the server, TiddlyWiki the client. This achieves separation of concerns; portability and scalability; and most importantly, independent evolution. TiddlyWeb concerns itself with storing data while TiddlyWiki handles interaction. Different TiddlyWiki implementations can present different user interfaces for access to the same data. At core TiddlyWeb is very simple, because it concerns itself with only the task of storing and providing tiddlers. This simplicity makes it easy to scale the server. TiddlyWeb and TiddlyWiki operate across a highly constrained and well-defined interface (the HTTP API) meaning the server and client can (and do) evolve separately from one another.

Of course, there is no requirement that the client be TiddlyWiki. Anything capable of HTTP can interact with the data stored in a TiddlyWeb server. TiddlyWeb is essentially a specialized web server.

2.2.1.2 Stateless

As just a web server, TiddlyWeb maintains no session data for clients. This means that a single TiddlyWeb instance can be scaled across multiple server instances that present themselves as one. Caching and load balancing are easy to achieve with simple, non-invasive technologies. For most applications of TiddyWeb thus far this has worked well. While there have been occasional requests for server side session management tools, these requests have usually turned out to be from people who are not aware of alternatives.

2.2.1.3 Cache

TiddlyWeb's primary resources, tiddlers, may be effectively cached both in browsers and intermediary cache servers using strong ETags (note, however, the caveats about ETags below). On a busy TiddlyWeb server, the vast majority of the content can be serviced from cache. However, because tiddler content is frequently edited, headers are set to require cache validation per request.

2.2.1.4 Uniform Interface

TiddlyWeb's resources are accessible via URIs. HTTP clients make GET, PUT and DELETE requests of the resources. This means the clients of the information resource can be anything that supports HTTP. The server makes no demands on, and has no expectations of, the capabilities of the clients. TiddlyWeb clients can use resources as they need. This model is maintained within the TiddlyWeb code as well: between functional portions of the code, APIs have very low surface area. Constraint at the interface between sections means sections can improve independently.

2.2.1.5 Layered System

TiddlyWeb's use of HTTP provides the surface layer between clients and the resources held by the server. The client cannot see under the covers to make special actions. This helps keep the server simple. Internal to TiddlyWeb, the same model is followed. Interactions between request handling code and serialization and store handling code are constructed such that the implementation of a serializer or storage system is invisible to the request handling code. This means new serializers and storage systems may be created and updated as needed.

2.2.1.6 Code-On-Demand
One of the most fascinating features of TiddlyWiki, the tool that birthed the tiddler concept, is that tiddlers may be content or code. As code they are plugins, modifying the functionality of the TiddlyWiki. TiddlyWeb preserves this code-is-content, content-is-code hybrid while adding on-demand behavior, delivering tiddlers containing JavaScript where needed. There are security concerns in this context: ensuring that code used is the code desired and overcoming cross-domain browser constraints when utilizing code from multiple servers.

These architectural styles that lead to REST result in three categories of architectural elements.

2.2.1.7 Data Elements
TiddlyWeb presents data elements–resources–over its API, not processors or procedures. These resources are identified by stable identifiers. A client asks the server to GET a representation or to PUT a representation of a resource. Processing and rendering happens in the client. This proves flexible, allowing for complex clients without requiring complex handling in the server. TiddlyWeb, however, does provide a simple mechanism for extending the representations of a resource it is able to provide. In an ideal setting, client code would be able to render generic representations (such as JSON or well structured HTML) to multiple representations, but in practice users have preferred that the server handle this.

2.2.1.8 Connectors and Components
TiddlyWeb follows the architecture of the web with regard to the connector and component elements described by REST. It extends the model by structuring client, server and caching interactions within the server code using similar uniform interface and layered system constraints. This makes TiddlyWeb itself (the server) easier to scale, extend and test.

2.2.2 Development
TiddlyWeb has been developed iteratively with a worldwide community of developers and users. The development process has exposed wisdom for developing and designing networked tools and exposes some of the expectations that users and developers have for web services, especially ones that make claims to being "RESTful". These discoveries are based on anecdotes from the TiddlyWeb community, and though not conclusive, may be useful for other systems.

2.2.2.1 Web Wisdom
TiddlyWeb's efforts to implement the REST architectural style confirm what by now ought to be considered, due to the success of the web, common wisdom about networked information services. Putting information on the open web makes it accessible by a diversity of clients for a diversity of purposes. To maximize reference and reuse, names of resources need to be stable [1]. TiddlyWeb tiddlers have canonical URIs that are stable for the lifetime of each tiddler. URIs are sometimes names. When making names that are usable by humans, namespaces are a useful tool for avoiding collisions. In TiddlyWeb, bags provide tiddler namespaces.

Authentication and authorization on the web are complex: hard to get right, easier to get wrong. Efficiencies can be found by grouping permissions into manageable chunks with few but strict constraints. In TiddlyWeb, the bag provides the locus of access control, not the individual tiddler. An agent can read, write, create or delete in the context of a bag, so an access policy can be set for a suite of tiddlers in a single step. Not only is this cognitively easier to manage in practice it is also easier to manage in code. Similarly the answer to authorization (can authentic agent X access resource Y for action A?) should be "yes" or "no", not a spectrum between "yes" and "no". If a spectrum is required then it is likely that the resource model in the system design is not a good match for the activities users wish to perform.

2.2.2.2 Web in Practice
TiddlyWeb development and deployment highlights opportunities and difficulties with the modern HTTP-based web. Most glaring is that there are large gaps between how content-negotiation is specified [3] and how it is implemented and used. In practice, to get useful results, some adjustments must be made. For people who are exploring via a browser, the simplest way to view alternate representations is by appending a meaningful extension or query parameter to a URI[9]. This method is required because browsers provide no easy affordance for suggesting a desired representation. URL munging is quite useful but leads to a profusion of URLs for the same resource.

When developing, content type problems can be encountered as well: some HTTP client libraries make it more difficult than it should be to set an Accept header (see, for example, jQuery[10]). Sending a simple, single-option Accept header, one that explicitly states "I want representation X" is most common. That is, using Accept as a statement of a requirement, rather than as a suggestion to the server of possibly acceptable options. It is useful to create and use arbitrary new content types (TiddlyWeb uses text/x-tiddlywiki for some representations). This is most useful in a well known or well documented system, but is limiting in a situation where discoverability is required; however, the above limitation in servers and clients exist. In day to day use it appears that TiddlyWeb users do not require XML for their structured data. They prefer JSON. Adding XML support to TiddlyWeb's collection of default or optional serializers would be simple, but in two years no one has asked for it.

Authentication handling is hard to make fit well into the REST constraints. Cookies are a commonly preferred method for indicating authentication state, but they make responses which otherwise might be equal, different.

Browsers have gaps in their support for HTTP that require workarounds. Some versions of Internet Explorer are unable to see an ETag in the response to a PUT request performed from JavaScript, making edit handling more complex. Browsers have varying support for Vary headers [4], limiting flexibility when attempting to make resources cacheable. Web servers, notably Apache make it very difficult to host resources with encoded '/'

[9] Compare: http://hoster.peermore.com/recipes/wsrest/tiddlers/Lessons.txt and http://hoster.peermore.com/recipes/wsrest/tiddlers/Lessons.json

[10] http://api.jquery.com/jQuery.ajax/

(%2F) in URLs[11]. Hacks of the environment presented to server code are required to allow '%2F'. TiddlyWeb uses one called pathinfohack[12].

ETags [3] are complex to use correctly, but extremely useful if done right. It should be possible to Vary the cacheability of an ETagged resource by cookie and content-type headers, but this is not reliable because of browser difficulties. A solution used in TiddlyWeb is to encode content-type and user variability into ETags. This raises the complexity of server code.

"[H]ypermedia as the engine of application state" is a core principle of REST [2, 8] that is valuable in practice but can increase server complexity when generating representations. On systems with a small number of resources and representations such as TiddlyWeb, it has not been a requirement for successful use. TiddlyWeb's core HTML representation provides linking between resources by taking advantage of linking functionality in HTML, however the two other default serializations (text and JSON) do not and the linking is not missed. There is a clear cost with this lack: clients need to be aware of more "well known" URIs to perform actions. TiddlyWeb has fifteen. These are learnable via the HTML representation and system documentation. The hypermedia constraint may be more useful in the context of discoverable APIs, less in the context of an application and API that is being developed with open server code and open client code. Discoverability may be useful in systems where automatic integration is desired.

2.2.2.3 Software Design
TiddlyWeb takes advantage of software systems and designs that can be usefully applied to other open web based services. Most important to TiddlyWeb's development has been WSGI[13], Python's Web Service Gateway Interface. The interface guarantees a consistent environment across a variety of web servers, leading to easy deployment. WSGI has been so successful in the Python community that similar projects have started in other languages, including Rack[14] for Ruby and PSGI[15] for Perl. The stable interface encourages a compositional approach to development (mirroring the web) that eases comprehension, testing, and extension.

Interfaces and composition are useful throughout the entire application stack. Effective serialization from internal representations of entities to external representations of resources and effective storage of entities is best managed by subsystems that operate as a black box. For this to work a server side entity should not be responsible for its own serialization and persistence. Instead the entity should be passed to tools which provide the serialization and persistence functionality. Passing between systems and layers in code mirrors RESTful design, where a client talks to a server that might talk to another server that might talk to another server.

2.2.2.4 Community Response
Developers and users of TiddlyWeb have expectations for how a web API should operate and their diverse use environments expose problems with hewing too close to the REST constraints. Developers of client side code can be frustrated by browser and server restrictions that prevent the use of the PUT or DELETE HTTP methods. To work around those restrictions a methodhack plugin[16] was developed which allows tunneling of those methods over a POST.

Developers who are accustomed to atomicity during rename operations or when requesting or updating large numbers of entities do not like making multiple requests. Developers often want the server to provide actions which are by design constrained to the client. These actions include updates to or retrieval of just parts (rather than all) of a resource, or server side processing of two resources (for example comparing resources). User comments have shown there are conflicts between learnable, grammatically sensible URIs for resources and pretty ones that people like to share with their friends.

3. CONCLUSIONS
TiddlyWeb's name was a good choice. By mirroring the design and constraints of the web the TiddlyWeb system achieves excellent scalability and flexibility while still serving its core purpose. The lessons learned from the development of TiddlyWeb and feedback from developers and users can be distilled into a short list of general advice:

- When modeling interaction, use the REST architectural style not only at the level of the web, but also at the level of software components and modules within the server. This helps to keep concerns separate, which allows for independent evolution and extension of components. Use web server interfaces such as WSGI to help bring the layered style into internal server code.
- Where necessary or useful, bend the constraints of REST to achieve success in the face of other constraints. In TiddlyWeb these adjustments are usually in reaction to browsers not making things easy (e.g. content negotiation and full use of the uniform interface) or in reaction to social processes being more effective (at least initially) than the constraint (e.g. hypermedia as the engine of application state can be harder to manage than simply telling users about the available resources).
- When writing code, separate web request handling from the data model from the serialization model from the storage model from the authentication model from the authorization model. In each distinct section ask what are the resources and what are the simple REST-like actions that will be performed. Minimize state. Maximize separation and layers.
- When complexity must be introduced and a choice can be made between requiring complexity in the server versus requiring complexity in the client, choose the client. When the server is made more complex it is likely to require all

[11] Apache can be configured to accept encode hashes with: http://httpd.apache.org/docs/2.0/mod/core.html#allowencodedslashes but this does not in itself solve the problem.

[12] http://pypi.python.org/pypi/tiddlywebplugins.pathinfohack

[13] http://www.python.org/dev/peps/pep-0333/

[14] http://rack.rubyforge.org/

[15] http://search.cpan.org/dist/PSGI/PSGI.pod

[16] http://pypi.python.org/pypi/tiddlywebplugins.methodhack

clients to change, whereas one client changing to add functionality has no impact on other clients.

- When developing systems that allow frequent editing of resources, use ETags to handle edit conflicts [5]. Require cache validation for those resources that are edited often.

If TiddlyWeb were to be developed again, some things would be done differently:

- Instead of using extensions (e.g. *.txt) on URIs for faking Accept headers, query parameters would be used instead. Extensions are hard to manage when the extension is also a part of the canonical resource name (compare /bags/foo/tiddlers/bar.html and /bags/foo/tiddlers/bar.html.html).
- Greater use of links between resources in representations other than HTML. For example a JSON representation of a bag would include a tiddlers_uri key whose value would be a URI of the tiddler collection associated with the bag.
- All resources, not just tiddlers, would get strong ETags to enable better caching and more robust concurrent editing.

4. ACKNOWLEDGMENTS

TiddlyWeb is an open source project built under contract with Osmosoft[17], the open source innovation arm of BT. The code is copyright The Unamesa Assocation, "a non-profit, world-wide association of individuals from industry, academia, and NGOs that provides free software tools and web services for schools, clinics, and other community organizations"[18]. Many thanks to Osmosoft, BT, Unamesa and especially the growing TiddyWeb community. Thanks to Frederik Dohr and Walt Woolfolk for reviewing drafts of this paper.

5. REFERENCES

[1] Berners-Lee, T. 1998. Cool URIs don't change. http://www.w3.org/Provider/Style/URI

[2] Fielding, R. 2000. Architectural Styles and the Design of Network-based Software Architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[3] Fielding, R., Gettys, J. Mogul, J., Nielsen, H., Masinter, L., Leach, P., Berners-Lee, T. 1999. Hypertext Transfer Protocol –HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html

[4] Keays, R. 2007. Internet Explorer meets the Vary: header. http://www.ilikespam.com/blog/internet-explorer-meets-the-vary-header

[5] Nielsen, H., LaLiberte, D. 1999. Editing the Web: Detecting the Lost Update Problem Using Unreserved Checkout. http://www.w3.org/1999/04/Editing/

[6] Ruston, J. 2004. First version of TiddlyWiki. http://www.tiddlywiki.com/firstversion.html

[7] Socialtext, Inc. 2006. Socialtext REST Documentation. http://www.socialtext.net/st-rest-docs/

[8] Williams, P. 2007. Hypermedia as the Engine of Application State. http://barelyenough.org/blog/2007/05/hypermedia-as-the-engine-of-application-state/

APPENDIX

TiddlyWeb is a suite of Python packages that can be installed to create a web-service that hosts tiddlers: small, document oriented pieces of information and code. The tiddlers are contained in bags which provide user manageable concept and authorization domains. Bags are listed in recipes to produce collections of tiddlers for practical uses. TiddlyWeb has a main website at http://tiddlyweb.com/. This paper is not meant to be an overview of TiddyWeb, but rather the lessons learned from it. However, a quick demonstration of TiddlyWeb functionality can be had on a Python equipped machine by installing the Python package called tiddlywebwiki and starting up the built in server. One set of instructions for doing this follows. These instructions work on a system that has pip[19]:

```
pip install -U tiddlywebwiki
twinstance tiddlyweb
cd tiddlyweb
twanager server &
# In a browser go to http://0.0.0.0:8080/
# and browse around.
# To see a sample tiddlywiki go to
# http://0.0.0.0:8080/recipes/default/tiddlers.wiki
```

TiddlyWeb code is hosted at http://github.com/tiddlyweb/tiddlyweb

[17] http://osmosoft.com/

[18] http://unamesa.org/

[19] http://pypi.python.org/pypi/pip