wq: A modular framework for collecting, storing, and utilizing experiential VGI

S. Andrew Sheppard
University of Minnesota and Houston Engineering, Inc.
Minneapolis, MN, USA
sheppard@umn.edu

ABSTRACT

We present "wq", an open-source framework for developing robust applications allowing volunteers to collect geographic information (VGI) in the field. Successful VGI applications been deployed in various contexts, but much of the effort that is put into common programming tasks cannot be re-used, often because the application code is too closely tied to the problem domain. User-friendly campaign authoring tools are being explored as ways to facilitate the rapid deployment of VGI applications, but many of these tools necessarily enforce a restricted vocabulary of interface elements and database models, limiting their usefulness for more complex VGI project workflows. In contrast, we propose a highly modular, open-source approach that enables reuse of general-purpose components created to facilitate common design patterns – without enforcing any hard limitations on the database model or interface. The framework builds off of and extends numerous existing open-source projects and leverages open standards (e.g. HTML5), which means it will work across all popular mobile devices as well as desktop browsers. The ideas behind wq arose from our ongoing efforts to generalize an existing data collection application initially created for community-based stream quality monitoring. In this paper we justify the design decisions made in creating wq and suggest general principles that should be taken into consideration when designing systems for collecting, storing, and utilizing VGI.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Frameworks; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work

General Terms

Design, Human Factors, Languages, Standardization

Keywords

citizen science, crowdsourcing, HTML5, open source, VGI

ACM SIGSPATIAL GEOCROWD'12 Nov. 6, 2012. Redondo Beach, CA.
 ISBN 978-1-4503-1694-1/12/11...$15.00

1. INTRODUCTION

As the reach and breadth of the World-Wide Web continues to expand, traditional sources of authoritative information are increasingly being complemented with information contributed and maintained by non-professionals with varying levels of expertise. The field of GIS is no exception, and volunteered geographic information (VGI) is expanding as a research topic of interest[2]. Prominent examples of VGI platforms include Open Street Map [3] and Ushahidi [10]. As GIS plays a critical role in many community decision making processes, the "democratization of GIS" has brought many new issues to light. Some of the research questions include: how can we measure the quality of VGI [12]? How can we understand the motivations for participation in VGI projects? How can we make sense of the large amounts of geographic data already being shared online – and what constitutes "volunteered" GI?

Underlying most VGI research is the need to build or leverage computer systems for collecting VGI. It is difficult for smaller organizations and projects to build VGI systems, and once funding runs out or key individuals leave it is difficult to maintain them [1]. For researchers it is a challenge to balance the need to create engaging systems with getting research done. To address these issues, we introduce `wq`[1], a modular framework for building and deploying VGI systems. In this paper, we first categorize existing systems and related work, discuss the rationale and design decisions behind `wq`, and conclude by arguing for a standards-based open-source approach to building VGI applications.

1.1 Descriptive VGI: Delineating the world

The many forms of VGI can be broken into two broad categories with differing goals, approaches, structure, and uses. In general, VGI projects exist either to *describe permanent features* in the world, or to *monitor changes and events*. In the absence of existing parallel terms for these categories, we will refer to them as *descriptive* and *experiential* VGI, respectively.

A common application of VGI is to describe places in the world, and perhaps the most well known such system is Open Street Map [3]. Applications for descriptive VGI facilitate the generation of comprehensive geographic datasets through an iterative process of expanding and refining shared contributions.

[1] http://wq.io

Approaches. Systems for collecting descriptive VGI are often built as wikis. For example, Cyclopath [16] is a localized geowiki for cyclists in the Minneapolis/St. Paul metro area. The wiki approach is useful because it allows continuous improvement and review of contributions by other users. Revisions to the dataset are typically preserved in a history log, allowing mistakes or malicious edits to be reverted. Interfaces for contributing descriptive VGI are usually (though not always) map-based.

Data Structure. Descriptive VGI is typically made up of geometric features (polygons, lines and points of interest), together with a set of attributes and descriptions. Projects like Open Street Map and Cyclopath also include connectivity information on collected features to facilitate the generation of turn-by-turn directions. Other projects like Wikimapia place a heavier focus on a free-form textual description of the features.

Uses. In areas where the official sources of geographic data are limited or expensive, descriptive VGI can fill in the gap. VGI also serves as a valuable source of local knowledge such as place names that may not be familiar to those outside of an area or "unofficial" bike paths in the case of Cyclopath.

1.2 Experiential VGI: Monitoring changes

A second broad application of VGI is more temporal, allowing individuals to describe events as they happen, or to monitor long-term environmental changes. Eliciting experiential VGI is often referred to as crowdsourcing, though the term has applications in other domains. The Ushahidi platform is a prominent example in this category, and allows individuals to submit reports during crisis events [10]. Citizen science projects such as eBird [15], Creek Watch [7], and River Watch [12] allow volunteers to submit observations about the environment.

Systems in this category have a wide range of goals. Wiggins and Crowston [17] identify a number of broad categories of citizen science projects, including conservation, investigation, and education. Crowd-sourcing projects like Ushahidi have a goal of accountability, by ensuring that multiple voices are being heard.

Approaches. Mobile applications are particularly promising for experiential VGI, as they allow field-based entry of observations[4][6]. Other projects are concerned primarily with finding ways to use the VGI already being shared on the Internet. For example, Twitter allows tweets to be geotagged, and a number of projects are examining Twitter as a potential source of crisis information [14]. These categories are not mutually exclusive, and Ushahidi includes both a mobile interface for submitting data directly as well as ways to pull data in from Twitter.

Experiential VGI projects are relatively less likely to use a "pure" wiki approach. Instead, there is often a gatekeeper that reviews the data releasing it back to the public [15]. Further, because of the temporal and personal nature of the data, individual records cannot easily be improved by others if they are incorrect. Data quality is often addressed by looking for trends and discarding outliers, or by defining strict protocols for those collecting the data [12].

Data Structure. Experiential VGI generally does not contain geometry other than simple point features with timestamps and attributes. Depending on the project, the attributes collected may vary from simple textual observations and photos (e.g. Ushahidi, Creek Watch) to a series of predetermined parameters with established collection procedures (e.g. River Watch). The data may therefore be highly structured or have very little structure at all, particularly if it is being pulled from outside sources such as Twitter.

Uses. Experiential VGI is often used to complement authoritative data sources when funding is limited, and serve as input into community decision-making processes. In projects like Ushahidi, VGI has the potential to provide an alternative source of information in cases where "official" sources are not trusted.

Experiential VGI is generally much more topical than descriptive VGI, limiting the usefulness of general-purpose VGI applications like Open Street Map. Further, the unique process management needs of individual projects limits the economies of scale necessary to make custom software development sustainable. This issue is being addressed in a number of ways, as discussed below.

2. RELATED WORK

Other researchers have been concerned with the re-usability of systems built for VGI. Ciaghi et al.[1] describe the problem in the context of information and communication technologies for development (ICT4D). They note that the lack of a structured, re-usable approach to software development causes many ICT4D projects to ultimately fail once the initial funding period ends. `wq` provides an open source framework that explicitly encourages re-usability, and we hope it will be useful in a variety of VGI contexts including ICT4D.

MoCoMapps [4] and Sensr [6] allow individuals to easily initiate mobile campaigns for the collection of VGI. These systems store project definitions in a shared repository with a predefined vocabulary of interface elements and field types. This approach makes it possible to start a project with little or no programming knowledge, but it also limits the usefulness for projects with complex work-flows and data management needs. In contrast, `wq` provides a set of modules and tools that solve common application implementation tasks, allowing developers to focus on implementing the unique business rules specific to their projects. While this does require more programming knowledge, it also provides the flexibility necessary for more robust VGI applications.

Priedhorsky and Terveen [11] describe enhancements to the standard wiki database model necessary to make it more useful for VGI with more robust quality control needs. We provide tools for implementing common VGI database patterns, but intentionally avoid restricting wq-based applications to any set schema since the workflows are so different between projects. Manso et al. [9] summarize the issues of interoperability in GIS, and note that technical interoperability in particular is key for incorporating VGI into other GIS systems. As discussed next, a key factor in the initial motivation for building `wq` was the changing technical interoperability requirements of the projects we work with.

3. BACKGROUND

River Watch is a community-based monitoring program in the Red River basin, led by the International Water Institute[2]. Since 1995, teams of trained high school students and teachers have been recording water chemistry data and observations on field sheets. This information is reviewed by the program coordinators before being shared with the Minnesota Pollution Control Agency and becoming part of the official water quality dataset. The VGI management workflow for River Watch is explored in depth in previous work[12].

More recently, the success of the River Watch program has made it possible to expand into collecting other types of VGI, including macro-invertebrate counts, snow depth, and photos and observations while on canoe trips. In addition, a number of schools are now utilizing iPads as a learning tool both in the classroom and in the field, making mobile data collection a feasible alternative to the current paper-based method.

Eventually, the program is planning to expand beyond the traditional set of school-based volunteers. As part of a larger decision support system for the Red River basin[3], the IWI is looking for ways to allow anyone in the area to contribute real-time information on the ground during flood events in the valley. This will involve opening VGI tools to a wide number of contributors not traditionally associated with River Watch. Finally, the program is looking for ways to expand the lessons and tools of River Watch beyond the Red River basin to other watersheds.

The author has been working with the program since 2006, and previously built a custom database system for managing and maintaining River Watch water quality data [4]. As the program objectives started to expand, it was clear that the existing system would not able to meet these changing needs. However, it was also clear that implementing additional custom applications for each of the new VGI projects would be unsustainable and redundant. Instead, we designed a common framework with the following principles in mind:

- The code should be highly modular, so common components can be reused across systems. Existing technologies should be reused to the extent possible, and new projects should be able to contribute back to the core framework as they are built.
- The application(s) should be usable on a wide array of devices, including smart phones, tablets, and desktop computers - whatever volunteers are already using.
- The schema(s) should be flexible, so minor changes to the types of VGI collected do not require developer intervention.
- The system(s) should be capable of generating multiple output formats, for submission to various agencies and exploration in third-party tools (e.g. Google Earth).

[2] http://www.iwinst.org/education
[3] http://www.rrbdin.org
[4] http://riverwatch.umn.edu

As development began, it became clear that these principles could apply to many VGI projects, and that the code being written could be further generalized into an open-source framework for creating VGI applications – giving rise to `wq`.

4. DESIGN DECISIONS

The primary contribution of `wq` is its modular approach, with a relatively strict separation of concerns. Each part of `wq` is designed to do a single set of tasks and do it well. This is important for reusing pieces of `wq` in projects that are unable to utilize the whole framework, and indeed parts of the framework have already been deployed in various applications unrelated to River Watch.

As can be seen in Figure 1, the `wq` framework is itself composed of three relatively independent libraries, which facilitate the *collection*, *storage*, and *utilization* of VGI, respectively. The libraries are built to address specific challenges the author has seen in these areas. In the next sections we describe the design decisions made for each of the three libraries, and argue that VGI applications in general should use a similarly modular, open source approach.

4.1 Collecting VGI with wq.app

`wq.app` is designed to facilitate the rapid deployment of multi-platform applications for VGI, by solving common interface problems in a general, re-usable way. While the unique structure of each VGI application is left to the application developer, the goals of `wq.app` necessitated certain initial design decisions, as discussed below.

The increasing availability of smartphones and tablets is a promising trend for experiential VGI. Mobile devices come packed with sensors like GPS and cameras that can supplement the observations entered by the volunteer. The ability to record events as they occur can serve to increase the quality of the data[12]. VGI mobile applications can be designed to to work even when the device is not connected, and preserve the data until an opportunity to "sync" arises.

However, each mobile platform comes with a separate ecosystem of programming languages and interface elements. VGI application developers can easily end up spending a lot of time integrating platform-specific device APIs, knowledge that is not easily transferred to other systems and platforms. It is not uncommon for VGI projects to focus on a single platform of choice – thereby excluding a large number of potential volunteers.

In response, there are a few multi-platform solutions for mobile. We looked into using Appcelerator Titanium, which allows developers to produce "native" applications without needing to learn each OS stack. However, the abstraction is made possible through custom JavaScript libraries unique to the Titanium ecosystem. Similarly, Adobe provides a cross-platform solution with the Flex builder, but that requires an additional runtime to be installed on some devices. These solutions are arguably more reusable than a pure "native" approach, but merely shift the vendor lock-in problem around rather than eliminating it.

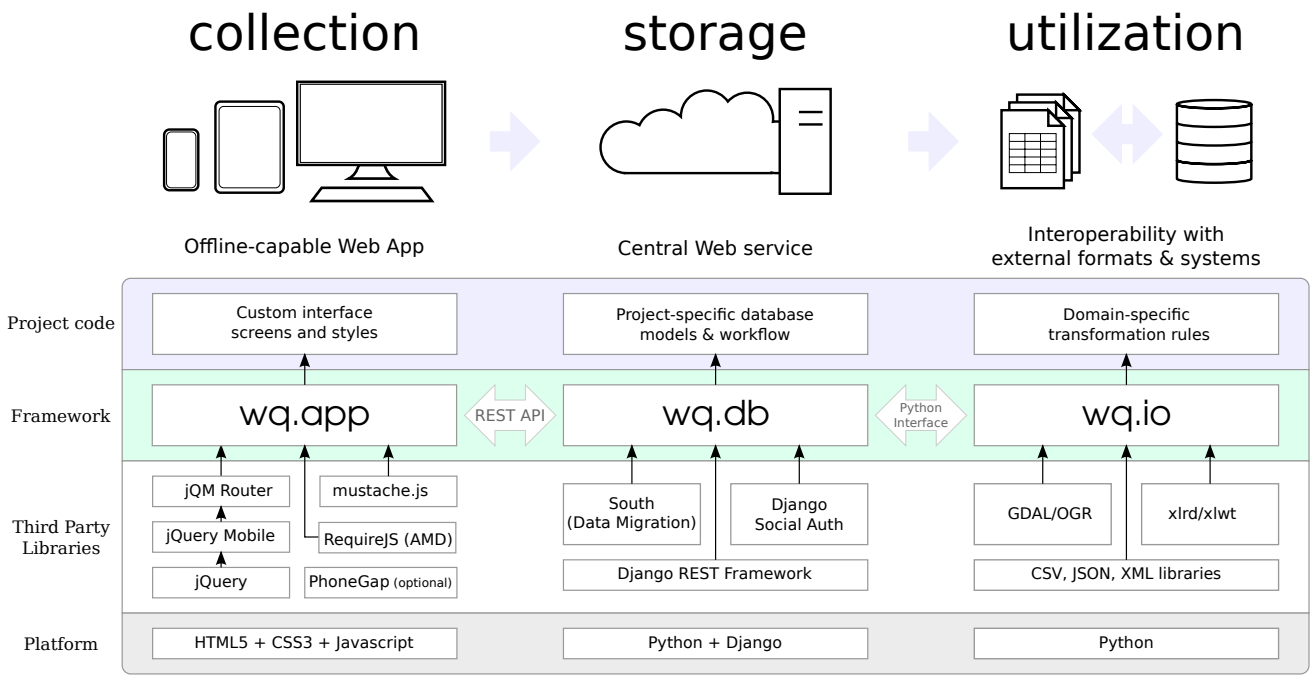

Figure 1: The structure of a wq-based application for VGI.

4.1.1 Cross-platform via HTML5

After looking into the alternatives, we elected to build `wq.app` as an HTML5 JavaScript library. HTML5 is a standards-based solution that fits very well with the needs and goals of `wq` and most – if not all – experiential VGI projects. Because nearly every smartphone and desktop is equipped with a web browser, HTML5 provides a natural cross-platform solution that does not require any additional software to be installed.

It is accepted that a web-based solution is preferred for submitting VGI on desktop computers. Nevertheless, while HTML5 has been noted as a promising avenue for VGI mobile application development [13], relatively few implementations to date have utilized it. The accepted view among developers seems to be that a "native" approach is preferred, so we will spend some time justifying our decision here.

Many of the requirements of VGI are addressed by features that had not been available in HTML until version 5. While much of the "buzz" around HTML5 has often focused on novel rendering techniques like the Canvas API and WebGL, the needs of VGI procurement are met by other features including offline storage and access to device sensors. These features still being standardized, but as can be seen in Table 1, the basic requirements for experiential VGI are already implemented in existing popular web browsers.

HTML applications can take full advantage of the established capabilities of the web, a fact that is surprisingly overlooked in the performance-focused "HTML5-vs.-native" debate. Web-based systems can be accessed from a desktop computer if needed; in our experience this is a core user requirement. Further, it is easy to rapidly iterate on prototypes by simply sending out a link to test, rather than needing to package, digitally sign, and request users to install the application. User-contributed VGI artifacts can be assigned meaningful URLs, making it easy to re-share them via existing social networks while preserving a central repository. CSS3 media queries can be used to automatically change the layout for different screen sizes and resolutions, a technique commonly referred to as responsive design. Finally, there exists a rich ecosystem of freely available JavaScript libraries that facilitate interface design and connectivity.

As a software platform, HTML5 is not without its limitations, and we experienced some issues when utilizing the new features in production applications. In particular, we had some challenges working with Web Storage, the only widely implemented offline data persistence API. Most browsers limit local storage to 5MB per website, which is effectively 2.5 million characters due to the use of UTF-16 in JavaScript strings. We were able to work around this by utilizing indexing and callbacks to reduce redundancies in the data.

In contrast, configuring the Application Cache to store the project resources offline was as simple as creating a text file (a manifest) with a list of files to cache. The API triggers a number of JavaScript events that can be used to notify the user when an updated version is available.

Table 1: VGI & HTML5

VGI Requirement	Current Solution[a]	Long Term
Offline application	Application Cache	
Offline data	Web Storage	Indexed DB
GPS capture	Geolocation	
Photo capture	File Input Type	Capture API

[a] Available in the latest versions of all popular desktop and mobile browsers. See http://caniuse.com.

GPS data can be obtained from most devices through the widely deployed Geolocation API. However, until recently it has not been easy to open the device camera to capture photographs and video. This is addressed in the latest versions of Android and iOS, and it is now possible to request a photo simply by including an `<input type=file>` in the document HTML.

However, these OS versions are still not available on every device, and it is easy to foresee new device capabilities not being made available to the web browser immediately. PhoneGap[5] (open sourced as Apache Cordova[6]) provides a workaround, making it possible access native device APIs from JavaScript via a custom "web view". While this adds some native functionality to web apps, debugging applications remains a challenge, as the native SDKs for each platform have limited support for debugging JavaScript code. To address this, the Cordova project provides a remote debugger (Weinre) that facilitates debugging for both web and "hybrid" (PhoneGap-based) apps.

Wrapping HTML5 applications in PhoneGap also makes it possible to distribute them on the respective markets for each platform. While there should generally not be a need to charge volunteers to download VGI applications, the platform marketplaces provide an established way to distribute applications until web apps become more familiar to users. The primary goal of PhoneGap is to promote the web as an application platform by providing access to features that have traditionally only been available to native applications, while working to get those features into HTML5. In this sense, "the ultimate purpose of PhoneGap is to cease to exist." [8]

We contend that the effort spent on building VGI applications using non-standard technologies due to perceived or actual deficiencies in HTML5 would be better spent improving PhoneGap, or the even the HTML5 spec and implementations directly. We acknowledge that existing projects will need to carefully weigh the costs of changing technologies, but we encourage aspiring new VGI projects to look to HTML5 as a first choice for their development platform. `wq.app` provides a tested, practical way to get started.

4.1.2 Modular code via AMD and RequireJS

As many web developers know, there is a trade-off between ensuring JavaScript applications can be downloaded quickly, and ensuring they are easy to maintain. This is because splitting JavaScript into multiple files means they must be downloaded individually with other web assets, increasing network traffic overhead. This trade-off can be alleviated through use of a build system, that allows developers to work with distinct modules during development and then build a single compressed file for deployment.

While we had previously used a custom build process in our applications, when creating `wq.app` we migrated to Asynchronous Module Definition (AMD), which is rapidly becoming a de-facto standard for modular JavaScript development. With AMD, each JavaScript module explicitly declares its dependencies and a module definition callback function that executes once the dependencies are loaded. Importantly, the dependencies can be either project-specific modules or external libraries. This contrasts with the traditional method of library use, which required developers to carefully organize the `<script>` references within their HTML files, to give library code an opportunity to set a global variable so they could use it in their application code.

During development, AMD loaders like RequireJS[7] resolve the dependency list and automatically download the required script modules. This facilitates debugging as each module typically has its own entry in the browser debugger's resource list. When the application is ready for deployment, the RequireJS optimizer combines the dependencies into a single file and "minimizes" the code, removing comments and renaming variables to reduce file size.

In keeping with the design principles of AMD, `wq.app` does not define a single global object for use by VGI applications. Instead, application developers can use the AMD API to request the specific `wq.app` modules needed for a particular project. `wq.app` also provides a build script that utilizes the RequireJS optimizer and supplements it with additional features, like automatically generating an appropriate Application Cache manifest for the built application.

4.1.3 Mobile-friendly interface via jQuery Mobile

There are a number of HTML+CSS+JavaScript libraries that facilitate mobile interface design. While we looked at several of these, we settled on jQuery Mobile as it is quite stable and served our goals well. In contrast with other similar libraries, jQuery Mobile prioritizes compatibility with established web development practices, works decently on desktop browsers (even Internet Explorer!), and for some applications can even work without JavaScript enabled. A surprising majority of the other libraries focus exclusively on mobile devices, and often limit support to the Webkit rendering engine used in Android and iOS. While acknowledging how frustrating it is to support traditional rendering engines, we argue that ignoring desktop browsers in favor of a mobile-only approach robs HTML5 of one of its greatest advantages as a universal application platform.

Another feature relatively unique to jQuery Mobile is the choice of a consistent interface design, rather than implementing look-alike styles for each popular mobile platform. We argue that providing a usable, consistently branded interface should take priority over going out of the way to convince users they are using a "native" app. That said, like any other HTML5 platform it is possible to swap out different CSS style sheets for each platform while keeping the application code the same.

To enhance the mobile experience while preserving existing website structures, jQuery Mobile comes with with a "hijax"[5] engine that captures clicks on regular HTML hyperlinks, fetches the HTML for the target page via AJAX, and smoothly transitions to the new page (optionally using a range of native-inspired animations implemented in CSS3.)

[5] http://phonegap.com
[6] http://cordova.io
[7] http://requirejs.org

The "hijax" approach is backwards-compatible: if the necessary JavaScript capabilities are not available, the browser will still navigate to the requested page using traditional methods. This overall approach works well for applications that always have online connectivity, but did not fully address the requirements of `wq.app`.

Instead, we synthesized a relatively novel approach that works well both offline and online, and whether the application is stored in the browser application cache or wrapped in a PhoneGap web view. We utilize the hijax capability of jQuery Mobile to intercept hyperlink clicks, but instead of fetching HTML from the server, we render the target page within the application before loading it. This makes it possible to handle arbitrary URL requests, often without any network traffic. The detailed process is broken into subtasks, each of which are handled by different `wq.app` modules and their dependencies.

`router.js` parses URLs captured by the hijax approach and ties them to pre-registered event callbacks. Most of the work is accomplished by leveraging the existing jQuery Mobile Router plugin.

`template.js` provides a simple way to manage and render HTML templates using the Mustache library[8]. The Mustache template syntax enforces a strict separation of form and function: it is impossible to insert arbitrary JavaScript code in a Mustache template. This separation of concerns makes it much easier to maintain projects in the long term, and it also makes it easy to use identical templates for rendering pages on the server, useful when webpages are accessed directly via traditional means.

```
<ul data-role='listview'>
{{#list}}
   <li><a href="/items/{{id}}">{{name}}</a></li>
{{/list}}
</ul>
```

Figure 2: A simple Mustache template

A Mustache template contains simple placeholders for attributes that are provided separately via a "context" object when the template is rendered.

`pages.js` connects `router` and `template` by providing a streamlined way to define callbacks that take a captured URL, create or load a context object, and render and display the finished page.

`store.js` manages the Web Storage, providing a simple query syntax for requesting objects from the store. If objects are not in the store, they are fetched automatically from a webservice or REST endpoint like `wq.db`. The returned JSON objects provide a context that can be used in the renderer. Since objects often come in collections, `store` includes some efficient functions for partitioning collections and retrieving individual objects.

`app.js`, the highest level module, brings all of the above modules together. When a URL is captured by the router, `app` requests the collection from the `store` using the collection URL, selects an appropriate template, and renders the output page. `app` can even be configured to automatically load related objects from other collections, effectively resolving database foreign keys from within the client. `app` is initialized with a simple configuration object (including a REST endpoint URL) and a set of HTML templates, then takes care of the rest.

These optimizations make it possible to build robust, offline-capable applications for VGI. They also provide a significant performance boost: if all of the necessary data has been preloaded, the requested page will be rendered and displayed nearly instantly. Thus, we believe `wq.app` and the libraries it utilizes address most of the issues associated with building HTML5 applications for experiential VGI.

```
// main.js
require(['wq/app', 'config', 'templates'],
function(app, config, templates) {
   app.init(config, templates);
});
```

Figure 3: Initializing a wq.app-based application

4.2 Storing VGI with wq.db

`wq.db` provides long-term persistence and curation capabilities to the VGI collected and submitted from `wq.app`. There are a range of approaches to solving the general issue of data storage and retrieval on the web. On the one hand are fully custom solutions that communicate directly with a backend database using direct SQL calls or a database-specific API. The original River Watch database fell into this category. As we have noted, this had negative implications on the ability to re-use the application code for other systems.

On the other end of the spectrum are comprehensive, pre-built content management platforms like WordPress and MediaWiki. These applications provide a ready-made solution for common content management tasks. While they can be extended, they each impose a single, predefined database model on all projects that use them. Somewhere in the middle are web frameworks, which generalize common database management tasks without enforcing a predefined database model.

4.2.1 Flexible design via Django and South

We opted to utilize the Python-based Django[9] web framework as the core of `wq.db`. Django comes with a built-in Object-Relational Model (ORM) that abstracts the task of generating database queries, allowing application developers to focus on defining the schema and application code. As shown in Figure 4, models (database tables) are defined in Django as Python classes, making them easy to extend and reuse. Django can even automatically generate a production quality administration interface directly from the model definitions. Importantly, the ORM includes spatial capabilities via GeoDjango, and can work with a number of backend database systems including PostgreSQL/PostGIS.

[8] http://mustache.github.com

[9] https://www.djangoproject.com/

Without the support of an ORM, any major change to the schema requires the manual definition of SQL migration scripts. In conjunction with the South extension[10], Django can automatically generate platform-independent schema migration scripts when the database model changes. South proved invaluable when migrating the River Watch database and server components to `wq.db`.

```
from django.contrib.auth.models import User
from wq.db.annotate.models import AnnotatedModel
from wq.db.identify.models import IdentifiedModel
from wq.db.relate.models   import RelatedModel

class Site(IdentifiedModel, RelatedModel):
    name = models.CharField(max_length=255)

class Event(AnnotatedModel):
    site = models.ForeignKey(Site)
    date = models.DateField()
    user = models.ForeignKey(User)
```

Figure 4: Extending abstract wq.db models in an example application

User account management in `wq.db` is provided by the Django's built-in authentication library, as well as by the Django Social Auth[11] framework, which streamlines the integration of third-party authentication mechanisms like OAuth and OpenID. Allowing volunteers to contribute VGI with their Google or Facebook accounts removes one common barrier to entry: the need to create and remember yet another online account.

4.2.2 *User-defined attributes and relationships*

In keeping with the principles of Django and `wq`, `wq.db` does not enforce any particular database schema, instead allowing developers full flexibility to define models and workflows appropriate for their VGI applications. However, `wq.db` does provide some predefined models and base classes that address design patterns we have come across repeatedly in building applications for VGI. These high level patterns are discussed below.

`wq.db.annotate`. Many applications of VGI ask contributors to go to a given location and record a set of parameters. Suppose the data points are being stored in an `Event` table with columns indicating location, timestamp, and contributor. If the specific parameters being measured are defined as additional columns on the `Event` table, new questions cannot be added without changing the database schema. This requires additional developer time for what is essentially a trivial change. Instead, it is much more flexible to store question definitions as rows in a separate table, and link the tables together with via a many-to-many relationship. This is much easier to maintain, at the cost of greater complexity. The `annotate` module helps facilitate this design pattern.

`wq.db.identify`. If the locations being monitored in a VGI project are pre-determined like they are in River Watch, they are commonly assigned unique identifiers by the project itself and potentially by one or more external agencies. The various identifiers can be defined as separate columns on a `Site` table, but this again puts an arbitrary limit on the number of identifiers that can be recorded. The `identify` module allows multiple identifiers to be assigned to each site. Each identifier can be associated with the authority that assigned it. If applicable, this information can be used to automatically generate links to an external authoritative webpage for the given site identifier. Identifiers can also be used within the site to make more meaningful URLs.

`wq.db.relate`. There are a number of possible relationships between monitored sites. In the case of River Watch, sites are upstream and downstream from one another, information that can be used to aid data exploration. Implementing these kinds of relationships can require dozens of many-to-many tables, requiring developer intervention whenever a new one is created. The `relate` model provides an alternative to this by making it possible to define new types of many-to-many relationships within the database.

It is important to reiterate that the `Event` and `Site` tables in Figure 4 are not defined anywhere in `wq.db`. This is intentional, to allow individual projects full flexibility in defining those models. This made for an interesting implementation challenge, as the `Annotation`, `Identifier`, and `Relationship` models need to have a foreign key pointing to the models they are describing. Django provides a workaround with the `contenttypes` module, which defines the concept of a "Generic Foreign Key" that is composed of both the object identifier and an identifier for the table (or "content type") it is stored in. The modules in `wq.db` utilize this feature to its fullest.

For interaction with VGI clients, `wq.db` comes with a built-in REST API, built on the Django Rest Framework[12]. When used with `wq.app`, this facilitates an intuitive one-to-one-to-one relationship between database records, REST URLs, and application screens. Utilizing the language-agnostic Mustache templates discussed previously, `wq.db` can directly render HTML screens when needed – for example to promote indexing by search engines.

4.3 Utilizing VGI with wq.io

While the collection of VGI has benefits for the participants, it is more valuable for everyone if the information collected is actually used. Descriptive VGI is often usable as-is – often within the system used to create it. Experiential VGI, on the other hand, is often used by combining it with data from other sources in traditional GIS systems. This is a particular challenge because the structure of the data within a VGI system often does not match that the requirements of those who would like to use it.

There have been a number of attempts to standardize data formats for various domains of VGI and traditional GIS. While this standardization work is important, it will take time to complete and will necessarily be restricted to certain types of GI. Instead of implementing specific protocols, `wq.io` facilitates interoperability with both current and

[10] http://south.aeracode.org/
[11] http://social.matiasaguirre.net/
[12] http://django-rest-framework.org

unanticipated formats by generalizing the process of converting and transforming data.

By providing a modular class system, `wq.io` abstracts the process of data transformation and file input/output, allowing VGI systems developers to focus on implementing the transformation rules specific to their domain. This is accomplished by abstracting the process of looping over a recordset to a consistent interface, regardless of the underlying file format and Python library used to parse it. The library provides a number of "mixin" classes that facilitate different transformation steps along the way.

5. CONCLUSION

A key factor in determining the success of computer systems for VGI is the long-term maintainability of those systems. Absent a critical mass of users and ongoing funding, many small custom VGI applications will not be sustainable. This can be addressed by generalizing across multiple applications and creating re-usable modules for VGI.

Web apps are a natural platform for the collection of VGI, and much of the work is accomplished via existing web frameworks like Django and jQuery Mobile. To meet the specific needs of VGI systems, we have introduced `wq`. Each project utilizing `wq` is able to contribute new features to the core libraries, thereby benefiting other projects. Because of the modular approach, systems designers can pick and choose which parts of `wq` to include in their project.

Like the frameworks it is based on, `wq` is freely available under a permissive Open Source license and can be downloaded from the project homepage[13]. It is our hope that other projects for VGI will be able to both use and contribute back to the `wq` library, with the flexibility to maintain control over their own project code. In general, we would like to encourage the field as a whole to gravitate toward open standards and open source software when building and deploying systems for collecting VGI. There is much left to be done, and `wq` is a step toward that goal.

6. REFERENCES

[1] A. Ciaghi and A. Villafiorita. Crowdsourcing ICTD best practices. In *ICST AFRICOMM '11*, pages 167–176. Springer, 2011.

[2] M. Goodchild. Citizens as sensors: the world of volunteered geography. *GeoJournal*, 69(4):211–221, 2007.

[3] M. Haklay and P. Weber. OpenStreetMap: user-generated street maps. *IEEE Pervasive Computing*, 7(4):12 –18, Dec. 2008.

[4] S. Hupfer, M. Muller, S. Levy, D. Gruen, A. Sempere, S. Ross, and R. Priedhorsky. MoCoMapps: mobile collaborative map-based applications. In *ACM CSCW Companion '12*, page 43–44. ACM, 2012.

[5] J. Keith. Hijax. http://domscripting.com/blog/display/41/, Jan. 2006.

[6] S. Kim and E. Paulos. A subscription-based authoring tool for mobile citizen science campaigns. In *ACM CHI Extended Abstracts '12*, page 2135–2140. ACM, 2012.

[7] S. Kim, C. Robson, T. Zimmerman, J. Pierce, and E. M. Haber. Creek watch: pairing usefulness and usability for successful citizen science. In *ACM CHI '11*, page 2125–2134. ACM, 2011.

[8] B. LeRoux. PhoneGap beliefs, Goals, and Philosophy http://phonegap.com/2012/05/09/phonegap-beliefs-goals-and-philosophy, May 2012.

[9] M.-A. Manso and M. Wachowicz. GIS design: A review of current issues in interoperability. *Geography Compass*, 3(3):1105–1124, 2009.

[10] O. Okolloh. Ushahidi, or 'testimony': Web 2.0 tools for crowdsourcing crisis information. *Participatory Learning and Action*, 59(1):65–70, 2009.

[11] R. Priedhorsky and L. Terveen. Wiki grows up: arbitrary data models, access control, and beyond. In *ACM WikiSym '11*, page 63–71. ACM, 2011.

[12] S. A. Sheppard and L. Terveen. Quality is a verb: the operationalization of data quality in a citizen science community. In *ACM WikiSym '11*, page 29–38. ACM, 2011.

[13] W. Song and G. Sun. The role of mobile volunteered geographic information in urban management. In *Geoinformatics, 2010 18th International Conference on*, pages 1 –5, June 2010.

[14] K. Starbird, L. Palen, A. L. Hughes, and S. Vieweg. Chatter on the red: what hazards threat reveals about the social life of microblogged information. In *ACM CSCW '10*, page 241–250. ACM, 2010.

[15] B. L. Sullivan, C. L. Wood, M. J. Iliff, R. E. Bonney, D. Fink, and S. Kelling. eBird: a citizen-based bird observation network in the biological sciences. *Biological Conservation*, 142(10):2282–2292, Oct. 2009.

[16] F. Torre, S. A. Sheppard, R. Priedhorsky, and L. Terveen. bumpy, caution with merging: an exploration of tagging in a geowiki. In *ACM GROUP '10*, page 155–164. ACM, 2010.

[17] A. Wiggins and K. Crowston. From conservation to crowdsourcing: A typology of citizen science. In *HICSS '11*, page 1–10. IEEE Computer Society, 2011.

[13] http://wq.io