CHAPTER 12

Making Your Web Site Mashable

This chapter is a guide to content producers who want to make their web sites friendly to mashups. That is, this chapter answers the question, how would you as a content producer make your digital content most effectively remixable and mashable to users and developers?

Most of this book is addressed to creators of mashups who are therefore *consumers* of data and services. Why then should I shift in this chapter to addressing *producers* of data and services? Well, you have already seen aspects of APIs and web content that make it either easier or harder to remix, and you've seen what makes APIs easy and enjoyable to use. Showing content and data producers what would make life easier for consumers of their content provides useful guidance to service providers who might not be fully aware of what it's like for consumers.

The main audience for the book—as *consumers* (as opposed to producers) of services—should still find this chapter a helpful distillation of best practices for creating mashups. In some ways, this chapter is a review of Chapters 1–11 and a preview of Chapters 13–19. Chapters 1–11 prepared you for how to create mashups in general. I presented a lot of the technologies and showed how to build a reasonably sophisticated mashup with PHP and JavaScript as well as using mashup tools. Some of the discussion in this chapter will be amplified by in-depth discussions in Chapters 13–19. For example, I'll refer to topics such as geoRSS, iCalendar, and microformats that I discuss in greater detail in those later chapters. Since I don't assume that you will have read any of those chapters, I will give you enough context in this chapter to understand the points I'm making.

Specifically, in this chapter, I will outline what content producers can do in two major categories:

* Ways in which they can make their web sites and content mashable without even producing a formal API
* Ways in which they can shape their API (features that are friendly to mashups)

Before content producers can decide how to act on any of this advice, they need to consider how remixability fits in with what they're trying to accomplish. We look at some of these issues first.

Tip For detailed notes on how to create, run, and maintain an API from the perspective of a seasoned API creator, please consult Chapter 11 of *Building Scalable Web Sites* (O'Reilly Media, 2006), written by Cal Henderson of Flickr.

Why Make Your Web Site Mashable?

To decide on how remixable you want to make your content, you need to understand what you want to accomplish. There is a wide range of interest with respect to making APIs. Some content producers (such as Amazon, Google, and Yahoo!) set out to develop a platform and therefore invest huge amounts of effort in creating an API. Others are interested in making things convenient for users of their content and create an API if it's not too difficult. Other content producers want to work actively against any remixing of their content. The course of action you take as a content producer will certainly depend heavily on your level of interest in the mashability of your content to others as well as the resources you have at your disposal to create an API.

Here are some arguments for why you might want to make your content remixable (in other words, why it's good not only for content users but also for you as a content producer):

* With a good API, developers and users can extend what you provide. Look at how the vast majority of the API kits for Flickr are developed by third parties rather than Flickr.
* Third-party developers can develop applications you haven't even thought of or are too busy to create. (I'd say geotagging is a huge example of this for Flickr—it opened up a whole new vein of activity for Flickr.)
* APIs appeal to users who are concerned about lock-in and want to use their content in places other than your web site.
* Many users are starting to expect to have APIs; as a result, having an API is a selling point to prospective users.
* With an API, you might be able to extend your presence and point others to you (for example, Flickr photos are distributed all over the Web, but they all link back to Flickr). Indeed, Flickr is *the* platform for photo sharing on the Web. But Flickr's attribution requirement (in other words, the photos served from Flickr need to link back to Flickr) keeps Flickr from being commoditized as a file-hosting service.
* If the API is of sufficient economic value to your users, it is possible to charge for using your API.
* In some cases, you might be able to create something like Amazon.com, which as a platform for e-commerce takes a cut of purchases built on top of its platform.
* Making your data more open is contributing to the common goals of the entire Web.

Using Techniques That Do Not Depend on APIs

Without creating a formal API for your web site, you can nonetheless make things friendly for mashups while creating a highly usable site for your users.

Use a Consistent and Rich URL Language

Chapter 2 analyzed the URL language of Flickr and showed how its highly addressable, granular, transparent, and persistent URL language opens up a lot of opportunities to mash up content from Flickr merely by exploiting Flickr's URL structures. The human-readable, transparent URLs of Flickr lets developers link deeply into the fabric of the web site, even in the absence of formal documentation. The fact that Flickr works hard to keep the URLs permanent allows mashup creators to depend on the URLs to keep working. Granular URLs give mashups very fine-grained access and control over resources at Flickr. You will learn in Chapter 14 how these same qualities make it possible to use a social bookmarking system such as del.icio.us to bookmark content from Flickr. Hence, developing your own web site with a rich URL language avails your content to similar mashup techniques.

Moreover, the discipline of creating a consistent and human-readable URL structure benefits you as a content producer. It forces you to abstract the interface of your application (for example, the URL structures) from your back-end implementation, thus making your web site more maintainable and flexible.

Use W3C Standards to Develop Your Web Site

The use of good standards helps bring clarity to your web design, especially standards that insist on separating concerns (such as content from design). For instance, disentangling formatting from the markup and sticking it into CSS has a side benefit for mashup folks of producing content that is clearly laid out. Even generating well-formed XHTML (instead of tag-soup HTML) would be a huge boon since it allows for more error-free scraping of data. All this makes things more parsable even in the absence of explicit XML feeds.

Pay Attention to Web Accessibility

An accessible site lets more people access your content. You might be required by law to make your web site accessible to people with disabilities (see `http://section508.gov/`). Even if you aren't legally obliged to produce accessible content, adhering to modern web design such as producing valid (X)HTML naturally contributes to producing better accessibility. The end product of increased accessibility (for example, clean separation of content from style) is more mashable than nonaccessible sites.

Consider Allowing Users to Tag Your Content

Tagging provides a lightweight way for users to interact with and label and annotate content. As I demonstrated in Chapter 3, those tags can be the basis of simple mashups. There are some tricky issues to consider when you create a system for tagging—for example, how to incorporate multiple words and what to do about singular vs. plural tags. There is no universally accepted way to do this, so you need to weigh the possibilities (I covered some in Chapter 3). Having a strategy for multilingual tags is helpful (in other words, how to handle Unicode).

Consider also whether you have built enough structure to allow the hacking of tags. Could a user have jump-started geotagging as was done in Flickr with your site? Do you have something equivalent to machine tags?

Make Feeds Available

In Chapter 4, you learned about syndication feeds, their syntax, and how they can be used to represent your content in different formats to be exported to other applications. Feeds are becoming ubiquitous on the Web—they're the closest thing to the lingua franca of data exchange. Users by and large are beginning to expect feeds to be available from web sites. Users like syndication; they spend more time away from your site than on yours. Feeds let people access data from your site in their preferred local context (such as a feed reader). Moreover, there is a whole ecosystem built around feeds. By producing feeds, your data becomes part of that ecosystem.

Creating feeds out of your web site should be very high on a priority list. In fact, depending on what systems you are using to publish, you might already be generating them (for example, weblogs or many content management systems). By virtue of pushing your photos to Flickr, YouTube, and many other social sharing systems, you have the option of autogenerating feeds.

Feeds sound intimidating, but don't worry. You can start small and grow them. You might have a single feed for the most recent content. See how that works for you. Then you can consider generating feeds throughout your system. (Remember that Flickr has an extensive selection of feeds.)

If you need to programmatically generate feeds, they represent a good place to start in the business of generating XML. You might ask which feed type to generate. Ideally, you should generate many types like Flickr does, which takes little effort. That is possible if you have an abstract model of the data that you then format for different format types by writing a template for each format. If you don't want to go through that effort, then Atom 1.0 is a good place to start. Atom 1.0 is now recognized by lots of feed aggregators. It's also a good stepping-stone toward building an API. (You would have the Atom Publishing Protocol, covered in Chapter 7, and GData as good prior art to start.) Moreover, Atom feeds can flow into Yahoo! Pipes and the Google Mashup Editor (GME). RSS 2.0 wouldn't be far behind in my priority list. Also, if you want to get a start on experimenting with RDF and the semantic Web, a good place to start is to produce RSS 1.0.

Let's return briefly to the issue of the feed ecosystem. As you have seen, Yahoo! Pipes and the GME use feeds natively. The Flickr API puts out many formats (as you saw in Chapter 6) but not RSS 2.0 or Atom, although there are many Flickr feeds. You saw in Chapter 11 that even with the extensive number of Flickr feeds to access the Flickr API, I still had to convert Flickr XML to RSS 2.0, which I did with Yahoo! Pipes. That conversion made the data available to the GME.

As a final note, try using feed autodiscovery to enable easier access to feeds by users (which was discussed in Chapter 4).

Finally, be friendly to extensions to feeds. Remember that RSS 2.0, Atom 1.0, and RSS 1.0 are all extensible. Make use of this extensibility. If your system consumes feeds that have extensions, don't strip them out.

Make It Easy to Post Your Content to Blogs and Other Web Sites

In Chapter 5, you learned about how blogs can be integrated with web sites such as Flickr. Flickr's Blog button allows users to post a photo to a weblog. Moreover, the Flickr All Sizes button makes it easy for users to embed a photo into a blog or other web site by providing HTML fragments that they readily copy and paste elsewhere. In a similar fashion, YouTube provides HTML to embed a video, and Google provides HTML to embed its maps and calendars. You as a content producer can emulate the practice of making it easy to post your content to other sites while linking back to your own web site, where the content originates. In addition to facilitating the flow of content from your web site, you track comments originating from other web sites through a variety of *linkback* mechanisms. (See Chapter 5 for more information.)

Encourage the Sharing of Content with Explicit Licenses

Licensing digital content clears away important barriers to creating mashups with that content. In your web site, you should allow users to explicitly set the licensing of content and data to use, such as the Creative Commons licenses do, for instance. Set defaults that encourage sharing, but always give your users the choice to change those defaults. Build functionality to enable users to search and browse content according to a license.

As you learned in Chapter 2, Flickr is a good model here. Flickr has done a huge amount to promote open content specifically licensed through a Creative Commons license. That users can explicitly tie a Creative Commons license to a piece of content has been a tremendous enabler for remixing. If you don't give a mechanism for your users to assert a certain license, there might be too much ambiguity around the reuse of content. Even if you don't have granular control over the licensing of content on the site, it's very helpful to have a global statement about intellectual property issues. That is, some content producers license an entire site in a certain way. For example, the Wikipedia is licensed under GFDL:

```
http://en.wikipedia.org/wiki/Wikipedia:Copyrights
```

Freebase is licensed under CC-By:

```
http://www.freebase.com/signin/licensing
```

In Chapter 2, we discussed the barriers to screen-scraping. If you don't have an API but don't mind your users accessing your data, consider creating some bot-friendly terms of service (ToS).

Develop Extensive Import and Export Options for User Content

The more ways you have to get data in and out of an application, the better. Ideally, you would support protocols and data formats that would help your users. As a bonus, let your users embed their data hosted on your site somewhere else on the Web (for example, through a JavaScript *badge*). Super-flexible badges can be used themselves to access data for mashups and can hint at the existence of a feature-rich API.

Study How Users Remix Your Content and Make It Easier to Do So

Be prepared to be surprised by how people might use and reuse your content. See how people are using your content, and make it easier to do so. The primary example I have in mind here is when people started to hack the Google Maps API. Google, instead of stopping those people, actually formalized the API.

At the least, if you don't want to develop an API, when you see people use your web site in unusual ways, you should think about what's really go on and whether to make it easier to carry out this reuse.

Creating a Mashup-Friendly API

Some web APIs are easier than others to use for creating mashups. In the following sections, I'll give advice to content producers aiming to make their APIs friendlier for consumption.

Learn From and Emulate Other APIs

You can learn a lot from studying what other API providers are doing. That's why this book is useful; you will learn about what API makers are doing—at least from the outside.

What are some great examples to study? Flickr is a good one obviously. Recently, I've come to appreciate the Google documentation as being really good too. It has a lot of copy-and-paste code, plenty of getting-started sections, and the API references. Often there are API kits in a number of languages; of course, a lot of time and energy went into creating this documentation.

Moreover, if you are a little player, consider making your API look a lot like those of the big players. For example, 23hq.com, a photo-sharing site, decided to mimic Flickr's API instead of developing its own:

`http://www.23hq.com/doc/api/`

That enabled Dan Coulter to support that API in addition to Flickr's API in `phpFlickr`:

`http://phpflickr.com/phpFlickr/README.txt`

If 23hq.com had built its own API, it would not be able to leverage the work of the much-larger Flickr development community.

Whether the creator of an API is flattered or irritated by the sincere imitation of the API by other players surely depends on context. Consumers of the API, however, will be all too happy to not have to learn yet another API to access essentially the same functionality from different web sites.

Keep in Mind Your Audiences for the API

You need to consider two distinct audiences when deploying a public API. The first is the direct audience for the API; this is the developer community, which includes those who will directly program against your API. The second is the indirect audience for the API but perhaps a direct audience for your web site: the possible audience for those third-party applications. Remember that although you have a direct audience in the developers, you are ultimately trying to reach the second, potentially much larger, audience.

Make Your API Easy to Learn

Good documentation of the features, the API, data formats, and any other aspect of the web site makes it much easier to understand and recombine its data and functionality. You should clearly document the input and output data expected. Do you provide pointers to schemas or ways to validate data? Documentation reduces the amount of guesswork involved. Moreover, it brings certainty to whether a function you uncover through reverse engineering is an official feature or an undocumented hack that has no guarantee of working for any length of time.

Why, for instance, do I recommend people using the Flickr API as a starting point (and maybe for the long term)?

* It's well-documented and has structures that make it easy to learn, such as the Flickr API Explorer at `http://www.flickr.com/services/api/explore/?method=flickr.photos`. (I don't know of any documentation for APIs that is as clear as this. You can try a query and see it happen.)
* It has lots of code samples.
* It has toolkits that implement the API in your favorite language. Flickr is ahead of the game here with more than ten language-specific implementations of the Flickr API.

The Flickr API Explorer is excellent and should be more widely emulated. It lets you invoke a method in the browser and see the response. The documentation lists not only the methods but also the input parameters and error codes. The great thing is that you can read the documentation and try something. Moreover, the Flickr API Explorer shows you a URL coming out of the REST API that you can copy and paste elsewhere.

Test the Usability of Your API

Use the techniques from Chapters 7 and 8 to remix your own site to see how mashable your site is and how well your API works. Review Chapter 11, and read in the feeds from your site into Yahoo! Pipes or the Google Mashup Editor.

You might be using your own APIs in an Ajax interface—but it's helpful to think like a mashup creator who is coming to your site for the first time and who will use more generic tools to analyze your site. It's interesting to see how your site looks from that point of view.

You can go further by extrapolating techniques from usability testing (`http://www.useit.com/alertbox/20000319.html`) to testing your API, instead of the UI of your web site. For instance, you could recruit a group of developers and give them a problem to be solved using your API. See what these developers actually do. Make changes to your API in response to feedback.

Build a Granular, Loosely Coupled Architecture So That Creating an API Serves You As Much As It Does Others

A public API for your web site does not have to be something you build only for others. Rather, it can be the natural outcome of creating a scalable and adaptable web site. One architectural pattern that has proven effective in creating such web sites—that of service-oriented architectures—is to decompose functionality into independent, fine-grained components (called *services*) that can then be stitched together to create applications. By defining clear interfaces among the services, one can change the internal workings of individual services while minimizing the effect on other services and applications that consume those services. It is this loose coupling of the components that makes the whole web site scalable.

With a set of granular services in place, you as a content producer have the building blocks of a public API. You can always start with a private API—which many Ajax interfaces demand. That way, you can decide to roll out a public API. (For instance, you use Firebug to study how the Flickr API is often being called by Ajax parts of the Flickr interface.)

If you decide to go for an API, make APIs an integral part of your site. The fact that the system depends on the APIs ensures that the APIs aren't just throwaway parts of the system. This provides assurance that the API is an integral part of the system.

For more insight into how service orientation benefits Amazon.com, which is a major consumer of its own services, read an interview with Werner Vogels, CTO of Amazon.com:

`http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=388`

Embrace REST But Also Support SOAP and XML-RPC If You Can

From Chapters 6 and 7, you know that REST is much easier for your users to get started with. (Amazon S3 in Chapter 16 provides another concrete case study of REST.) The use of REST and not just SOAP or XML-RPC lowers the barrier to entry. With REST, you can see results in the web browser without having to invoke a SOAP client (which is bound to be less available than a web browser). There's a strong argument to be made that by building a good RESTful human web API, you are already building a good API:

`http://blog.whatfettle.com/2007/01/11/good-web-apis-are-just-web-sites/`

However, if your primary developer audience is oriented toward enterprise development and is equipped with the right tooling, you might have to prefer SOAP/WSDL over REST. Remember that SOAP without WSDL isn't that useful. And if you do SOAP, be strictly observant of the version you are using.

Again, if you build an abstraction layer underneath, you might be able to handle multiple transport protocols. Your favorite programming frameworks might autogenerate REST or SOAP interfaces for your web application.

Consider Using the Atom Publishing Protocol As a Specific Instantiation of REST

If you build Atom 1.0 feeds, you're already on the road to building an API. Recall that there's plenty of prior art to be studied in the Google GData APIs if you want to go down this road. (Chapter 7 has a study of GData; Chapter 15 on the Google Calendar API is another study of GData.)

Encourage the Development of API Kits: Third Party or In-House

It's nice to have both the raw XML web services and the language-specific API kits. In theory, according to the argument of REST or the SOAP/WSDL camp folks, having the right web services should obviate the need for language-specific API kits. My own experience is the opposite. Sure, the Google GData APIs (see Chapters 7, 8, and 10) are RESTful, but having PHP and Python libraries is very useful. Even with WSDL, a language-specific API kit is handy.

Ideally you would have API packets for every possible language. Of course this is not practical—and not even the largest companies such as Google provide that many API kits. The priority is to have a good well-documented API. After that, I would say if you can put out an API kit in the language that you use in-house, that's already a great service. Google puts out API kits for its in-house languages. Microsoft puts out API kits in the languages it supports. Beyond that you need to talk to your potential developers and see what's important to them. (It's nice to have API kits that cover a range of languages.) Remember you don't have to develop all the API kits yourself—Flickr doesn't develop that many but provides a place to publicize them and promotes those API kits in the community of developers.

If you can provide both, it's nice to have a server-side language API kit and JavaScript API kit for client-side access.

Support Extensive Error Reporting in Your APIs

Note that for better or worse, it's very easy for developers to ignore error handling. You have to encourage them to handle errors. It starts with having good documentation of errors.

I'm of mixed minds about whether to embed the error in the XML body or as an HTTP response code. HTTP error codes are a standard way of dealing with errors, but it's not necessarily the easiest thing for new developers to understand. At any rate, in SOAP you can do fault handling in the fault code of the SOAP body. In XML-RPC, it's dealt with in the error body. I might suggest that even if you put error codes in the response body that you use the HTTP error codes as a starting point to build your own error response functionality.

Note The specification for the latest version of SOAP (1.2) now provides guidance on how to use the various 2xx, 3xx, 4xx HTTP status codes.[1]

Accept Multiple Formats for Output and Input

It's nice to have multiple ways of getting content in and out of an application. For example, Flickr has many ways to upload photos: the web interface, the desktop Uploadr, the API, and e-mail. Even so, some people have requested FTP and some type of mass downloading. Flickr doesn't offer FTP capabilities, but some people have worked to simulate it:

`http://blog.wired.com/monkeybites/2007/06/upload_to_flick.html`

For calendaring, you'll see the use of iCalendar and CSV in Chapter 15. In Chapter 13, you'll see how the proliferation of KML and geoRSS has been a boon.

Support UI Functionality in the API

As a consumer of APIs, I advocate support for all the elements available to users in the UI—and then some. It's frustrating for mashup creators to not be able to do something in the API that is clearly allowed by the user interface. There are sometimes good reasons to not enable certain actions in the API—but apart from such reasons, having a complete API is really helpful. As you saw in Chapter 6, there is a strong overlap between the capabilities of the Flickr API and the UI. There are some discrepancies between the API and UI—they got close but not an exact alignment.

Include a Search API for Your Own Site

You might consider adding an API to specifically enable searching of your web site. See Chapter 19 for how OpenSearch can then be used to integrate your web site's search functionality in other frameworks.

1. http://www.w3.org/TR/2007/REC-soap12-part2-20070427/#http-reqbindwaitstate

Version Your API

APIs, like all programming artifacts, are likely to change. Instead of having only one version of your API that can change, support multiple versioning of your API. That doesn't mean you will have to support every version indefinitely. Publishing a timeline for when you plan to retire a specific version of your API and documenting changes between versions allows the consumers of your API to make an orderly transition and adapt to changes in your API. Flickr doesn't explicitly version its API; for an example of an API with support of multiple versions, see the following:

`http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=118`

Foster a Community of Developers

A vibrant and active community makes a lot of mashup work practical. When making mashups, there are things that are theoretically possible to do—if you had the time, energy, and resources—but are practically impossible for you as an individual to pull off. A community of developers means that there are other people to work with, lots of examples of what other people have done, and often code libraries that you can build upon.

Don’t Try to Be Too Controlling in Your API

You should have a clear ToS for the API—see Chapter 6 for a discussion of the ToS for the Flickr API. Don’t try to be too controlling of your API. For instance, you might be tempted to forbid a user of your API from combining data from your site with that of other sites, such as those of your competitors. That’s very much against the spirit of mashups and is likely to antagonize your developers. I would argue that asking a user of your API to reference your web site is a good balance between the interests of the API consumer and API producer.

There are a lot of issues when it comes to establishing a policy for your API—but one is worthy of special consideration is that of commercial use. It’s not uncommon to make a basic distinction between the commercial and noncommercial use of an API, especially if you are not charging for the noncommercial use of an API. It’s useful to reflect on how Flickr and others handle the distinction in the context of your own business model. Remember, though, that it’s sometimes tricky to distinguish between commercial and noncommercial use; you will need to set up a process to make such a distinction.

Consider Producing a Service-Level Agreement (SLA)

A service-level agreement formally spells out the level of service a user can expect from a service and the remedies for failures to meet the expected level of service. It’s debatable whether most SLAs are of much practical use. What I really want is perfectly reliable service. Can any remedy offered by most service providers adequately compensate for disappointing that desire?

Nonetheless, as a user, I find that a thoughtfully constructed SLA reassuring because it gives me a sense of the level of reliability to expect from a service provider. A specific measurable target of performance is likely better than none at all. As an example, Amazon.com recently introduced an SLA for its S3 service:

```
http://www.amazon.com/b?ie=UTF8&node=379654011
```

Help API Users Consume Your Resources Wisely

Encourage the users of your API to consume compute cycles and bandwidth parsimoniously—most developers will want to cooperate. Document your expectations on the limits you set for the total volume or rate of API calls. Error messages from your API to indicate the throttling of API calls are very useful to consumers of an API.

When developing an API, it’s not unusual for you to issue keys to developers. However, tracking usage by the key alone is sometimes insufficient to manage the level of usage—keys are often leaked. You might track API usage based on a combination of key and originating IP address.

Both server- and client-side caching help with the performance of an API. You will want to help the users of your API to cache results properly. It’s extremely useful to have APIs that tell you when something has been updated and to return changes in the state of the data since a given time.

Consider Open Sourcing Your Application

If you want to open up your site to deeper remixability, you might even publish the source for your web site. Users will then have the option of studying the source directly should reverse engineering—or reading the relevant documentation—not give you the answers they need.

Easy-to-Understand Data Standards

The use of open data standards by content producers and consumers is a good thing, but it's hard for someone outside a field of endeavor to understand what those standards are and exactly how important they are. (For instance, it doesn't take a lot of time working with online calendars to grasp that iCalendar is an important standard, but it did take me some study to grasp how central it really is.) Hence, it is helpful if for every subject you could find a simple, clear articulation of the standards for a given field. In the absence of a clear consensus about what the relevant standards are, a trustworthy and clear-headed outline of the main contenders and the perceived strengths and weaknesses would be really helpful to an outsider or newbie.

The Cover Pages (`http://xml.coverpages.org/`) hosted by OASIS is the closest thing to such a resource that I've seen:

> *OASIS provides the Cover Pages as a public resource to document and encourage the use of open standards that enhance the intelligibility, quality, and longevity of digital information.*

Complementing a wide use of open standards is a concerted effort to generate API kits that comprehensively and accurately interpret these standards. For example, as you'll see in Chapter 15 in the discussion iCalendar, it's hard to tell how good any given API kit is at interpreting and creating that data format.

Moreover, the presence of good validators and schemas for any data formats would be extremely helpful to mashup developers. For example, the early days of working with KML were hard because there was so much trial and error with writing something and then feeding it to Google Earth to see whether it would work. With good validators in place, data producers can debug their data without less experimentation. Some examples of useful validators are as follows:

* `http://feedvalidator.org/`, which helps with KML as well as RSS and Atom feeds (remember Chapter 4)
* W3C validators (`http://validator.w3.org/` and `http://jigsaw.w3.org/css-validator/`) to check on the validity of (X)HTML and CSS, respectively

Summary

This chapter presented a series of techniques for making a web site more mashable. After explaining why content producers would want to make their data and services remixable, I then presented some techniques that do not depend on creating a formal API. The heart of creating a mashable web site is producing an API that is friendly to developers. I presented techniques for creating such an API, drawing from what you learned from the process of creating mashups in various contexts.