Web APIs

Introduction

According to a survey by Netcraft.com there are currently roughly 644,275,754 active web sites. Many websites are supported by a database backend. The site can communicate with the database to perform any number of operations, most commonly classified under CRUD (create, read, update, destroy). This is logical, but what happens when another application wants the ability to do CRUD operations to the aforementioned application? One way an application can communicate with another application is through a Web Application Programming Interface (API). Web APIs serve an important purpose to the Internet, they can be implemented fairly easily using a standard technique and they have a bright future.

Why?

The most obvious question to ask is why web APIs exist. The answer is simple, web APIs allow outside applications to perform operations on an application with an API. For example, say a program could send an API request to Facebook to get all status updates for a particular user. This application could then analyze this data and do any number of things with it.

Do's

Web APIs can be a very powerful tool if executed correctly. The most important aspect of a web API is the URL structure. The URL structure has to make sense. For example, it is logical to use a URL such as hospital.com/patients to get all the patients. This can get a little fuzzier when a URL has multiple variables. In the previous example, the want is for all patients. What if a doctor wants to see details for one of many emergency contacts for a patient? In this case, a URL like hospital.com/patients/:id/contacts/:id would be a sufficient structure.

It is also important to be a good citizen when creating a web API. This means gracefully handling bad data, bad requests, and bad responses. It is important to define a clear strategy to handling badness. A common way to handle bad data or bad requests is to return a success flag with every response. This way an application can expect and handle a bad response from an API.

Don'ts

Although a web API is pretty straightforward, there are a few things that need to be avoided when developing an API. It is important not to use more than two nested dependencies. For example a URL like hospital.com/patients/:id/contacts/:id/children/:id is not usable. In order for an application to use this URL it would need both the patient id and the emergency contacts id. It would be better to create a separate URL like hospital.com/contacts/:id/children/:id. Developers should also make sure they use

the correct "verbage" when creating an API. Make sure GET paths aren't manipulating databases and POST paths aren't trying to delete items from a database.

Security

Web APIs are inherently vulnerable in the aspect of security. Giving other applications the ability to manipulate a database sounds dangerous. Unfortunately, there is no security standard for creating an API. The best thing a creator can do is apply an API key to their API. An API key is requested by another application. It is a sort of identifier for the application and it is usually based on the system's UUID. Without a valid API key, another application will only receive errors when they try to access an API.

APIs can also require authentication before they will reveal data. One common technique is to use OAuth. OAuth requires clients to register themselves with an API before they can receive or send any data to the API.

How?

How does a web API work? A web API relies on REST (Representational State Transfer) requests to send and receive data. REST requests can fall under one of four categories: GET, POST, PUT, and DELETE. Typically POST and PUT requests are used to send data to a URL. A GET request is used to retrieve data from a URL and a DELETE request is used, well to delete. Web APIs usually send and receive data in the form of XML or JSON. JSON uses a key variable type notation and stores data in objects called JSON objects. An example Ruby JSON object is pictured below.

```
{"patient":[{"name":"Patient
1","age":35,"height":72,"weight":195},{"name":"Patient
2","age":23,"height":65,"weight":320},{"name":"Patient
3","age":44,"height":78,"weight":220}]}
```

Example API

Implementing a web API is fairly straightforward. First there has to be a need for an API. Commonly an API is needed to allow clients to modify a server's database. For this example, the server will handle medical information and the client will send requests to the server to create a new patient, update an existing patients record, delete a patient that has been released, and get a list of all the patients.

The first three requests all deal with a single patient. Therefore they can be handled under one URL using different actions. Let's say this URL is hospital.com/patients/:id where id represents the patient number. Doing a GET to this URL would retrieve all available information for that patient. This information could include height, weight, date of birth, medial allergies, and disabilities. A nurse

may use an iPad application to send a request like this to get to know their patient. Performing a POST to the url hospital.com/patients/:id would create a new patient with the id in the URL. Typically ids will be auto generated and require no programmer maintenance. If a new patient is admitted to the hospital, a front-desk person may enter the new patient's information and POST to the server. A nurse may be required to update the status of a patient (critical, stable, etc.). Performing a PUT to hospital.com/patients/:id will update the patient with the data sent. The data being sent to the server would look something like this (in Ruby).

```
{“id”: 1,"name":"Patient
1","age":35,"height":72,"weight":195}
```

The final request is somewhat different from the first three requests. This request wanted the ability to return all patients in the hospital. Instead of passing in an id, the URL will simply look like hospital.com/patients. Performing a GET request to this URL would return an array of JSON objects representing each patient in the hospital. A doctor may want to look at all the patients in their hospital so this is a crucial request.

The Future

As web APIs are relatively young, their future is bright. As more APIs are developed it is important to optimize them. This can be done with new API scripting languages like Webshell and Temboo. These languages streamline the API creation process and are designed for fast API consumption and implementation. Web APIs are often created to support mobile clients now. Tools to help build such APIs are now being developed. One of these tools is Tiggzi. Tiggzi is a mobile GUI development platform.

Conclusion

Web APIs are emerging as an amazing tool for web development. They allow simple and defined application communication in the web. If APIs are executed correctly they can be widely used, and if not they can fade away. Implementation is straightforward as long as a developer can follow API guidelines. Web APIs have not come close to reaching their peak, and in the future they will be lighter weight, more efficient, and more complex.

Reference:

http://news.netcraft.com/archives/2012/03/05/march-2012-web-server-survey.html

http://developer.shopping.com/kb/02n400000009R8wAAE/501400000009apNAAQ

http://www.apievangelist.com/2012/07/27/what-is-the-future-of-web-apis/