Contents

Manual testing vs Automated testing

Should we test our software using manual testing, automated testing or both?

Both manual and automated testing are crucial to developing a successful piece of software on time and in budget, so it's important to make them an integral part of your project's planning from day one. As the saying goes, a stitch in time saves nine—and with software testing, it's always best to plan for it early on rather than spend twice the money (and time) it would take to test on fixes and patches down the road.

You may choose to focus your testing efforts in one (or both) of two ways: manual testing or automated testing. Large applications may even require regression testing, which is designed to ensure that new changes don't break the old functionality that's already deployed. Each type of testing has its pros and cons—let's take a look at both in more detail.

Manual Testing

First, let's look at manual testing. Manual testing can be the more tedious and time-consuming of the two, especially if you're making lots of changes to your app in the future.

You're not paying for automation tools, but you are paying for someone's time—and in the short-term, that may be more cost effective. It all depends on what you have planned for the life cycle of your app.

What you lose in the efficiency of automation testing you'll get back with more real-world QA feedback like you'd get from the average user. In other words, you're getting the human element to your testing. That's because manual testing is just that—a tester going through your program and using it just like any other user would. By interacting with your app, a manual tester can compare expectations for how the app should run with outcomes, then provide feedback on what didn't work. They'll also be able to weigh in on the visuals—something computers can't do.

For instance, a manual tester could tell you that the contrast between a button and the background is too light, which makes it hard to see the button and understand what action needs to be taken. This type of user interface (UI) feedback is something automated testing wouldn't find, making manual testing closer to the sort of feedback you might hear from actual customers. If you have the time and a good QA team (larger applications require more than just one QA person), manual testing is the way to go.

Manual Testing Pros

Get visual feedback. Scripts can't provide opinions and input about how a UI looks and feels like a person can.

The human element. You're getting the exact kind of feedback a person would give you, and that can be invaluable. Being able to predict what your users will or won't like—things a computer can't give feedback on—ahead of time can influence your design and make it better from the bottom up.

Less expensive in the short-term. If you're only testing a simple app once, and don't expect lots of updates, manual testing doesn't require you to invest in expensive tools or software.

Flexible, on-the-fly testing. Testing requires you to write, program, and review test cases for your software. But, if you really only need to test one small change, you can manually test it right then and there.

Manual Testing Cons

Less thorough than automation testing. You always have to account for human error. When a script is running the testing, it's less likely to skip or miss things.

Testing fatigue. Just like how you wouldn't want the developers who've built an application to test it because they're so intimately familiar with it, there's the risk that your QA people could get used to your application and know how it works. This can lead to testing fatigue and errors slipping through the cracks. Tedious, repetitive tasks can make testers weary, so they're more prone to missing a mistake.

Not reusable. If you foresee a lot of changes and updates to your app in the future, you'll have to manually test all over again to ensure no new changes broke the build.

Automated Testing

Automated testing is all about comparing your expected results for how a program should function with how it actually functions. Automation testers will write and run tests that include a series of predefined actions that can occur in your software, then seeing if things run according to plan. When the outcomes of testing don't match the expectations going in, you've likely got some bugs, and automation scripts will point those out to your team so you can get them fixed.

Here's how it works: Automated testers setup testing scripts designed to go through various scenarios of a "user story." Every business requirement and functionality has a user story that defines how different users interact with the software.

Each page of your app or site, for example, could have several user stories. The scripts then go through every possibility of a user story and find errors where human testers won't. For instance, you could have a form where a user chooses one of 50 states. A script would go through and

submit a new user application for each of the 50 states to make sure they all work properly. A human tester would probably only submit an application for one or two states. Then, after a change or a bug fix is implemented, automated testing performs regression tests to ensure those new changes don't break old functionality.

Automated Testing Pros

If a bug is there, it'll catch it. Scripts can be much more thorough, catching bugs that humans might miss—due either to volume or monotony of the task.

Get the speed and efficiency of computers. Scripts are faster than humans, so you'll get more done in less time.

Reusable tests for code that gets frequent updates. If you're constantly making updates and republishing units, you don't have to rewrite test scripts each time, you can reuse them in your regression testing.

Gives your team a break. There's a better chance your development team will get fatigued and weary if they're manually testing an app after coding it. Automating testing will give them a chance to focus more on the big picture.

Better visibility into app performance. With automation testing, it's easier for the team as a whole to review results and be on the same page, vs. one person manually compiling testing results.

Automated Testing Cons

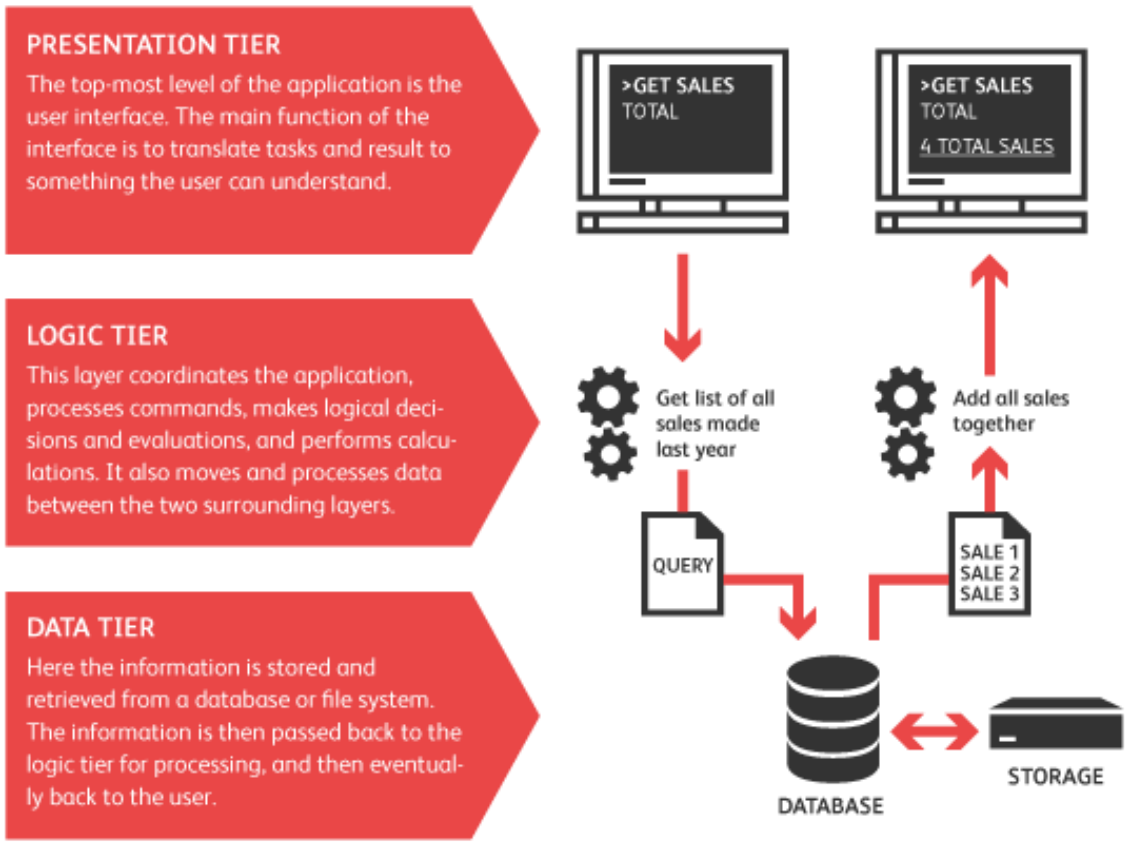
Lacks the human element. As we mentioned above, manual testing gives the added human element of an actual user interacting with an app, and all the preferences and visual cues that come along with it.

Less feedback. Without that human element, you also won't get insight into visual elements of your UI, usability or workflows.

Maintenance. Tests need to be maintained from time to time. If a UI element for example changes, then a whole test suite might require an update. Tests that aren't kept up-to-date might provide the tester with false-positives (a false impression that there's a bug in the system).

Application overview:

Let's start with a brief overview of the technology to try and clarify some of the misconceptions out there. Most Web applications lend themselves very well to the classic three-tier architecture model.



In this model, the entire application is separated into three parts: the data tier, the logic tier, and the presentation tier. Under ideal circumstances none of the tiers know anything about the platform, technology, or structure of any of the others.

Any one tier can easily be swapped out as new development (or test) needs arise. Further, it is quite common that each of these tiers runs on a different server, or in the case very large applications on different server nodes each with its own load balancers, firewalls, etc.

If this model is strictly adhered to, then test automation becomes simple: any of the tiers can be replaced with your choice of a test tool. If during development, the boundaries between the tiers start to blur, testing (as well as future enhancements to the application) becomes more difficult. This is one reason why you always want to get testing involved in the design process as early as possible.

The presentation tier is the UI. The presentation layer contains the components that implement and display the user interface and manage user interaction. This layer includes controls for user input and display, in addition to components that organize user interaction.

It's basically the visual representation of our software to the end user.

The logic tier is the API. This is where all of the business logic belongs, and if that is the case then this tier alone makes out the entire API. In certain cases, the logic tier of your application has more complexity, and it is not uncommon that multiple technologies (web server, message queue, et c) will make up this tier.

The data tier represents the Database. A data-tier application (DAC) is an entity that contains all of the database and instance objects used by an application. A DAC provides a single unit for authoring, deploying, and managing the data-tier objects instead of having to manage them separately. This is where all the information of our software (including the user's info) is stored on.

UI Testing

What is UI testing?

Graphical User Interface testing is mainly about ensuring that the UI functions in the right, that an app follows its written specifications and that defects are identified. Other than that, we check that the design elements are good. This involves checking the screens with the controls like menu bars, toolbars, colours, fonts, sizes, icons, content, buttons, etc. How do they respond to user input?

To do UI testing, we usually use various test cases (set of conditions that will help the tester determine if a system is working as it's supposed to) and there are 2 ways of conducting it; manually (with a human software tester) or automatically (with the use of a software program).

Capture & replay testing

We can also do the GUI testing by using some automation tools that were developed specifically for that. The idea is to run the app and capture/record the interaction that has to happen between the user and the app itself (mouse movements, etc.). We then repeat that for all the actions that users should have with the app and during the replay, those user actions that were saved will be reproduced and compared with what is expected.

Introducing Selenium

Selenium is a set of different software tools each with a different approach to supporting test automation. Most Selenium QA Engineers focus on the one or two tools that most meet the needs of their project, however learning all the tools will give you many different options for approaching different test automation problems. The entire suite of tools results in a rich set of testing functions specifically geared to the needs of testing of web applications of all types. These

operations are highly flexible, allowing many options for locating UI elements and comparing expected test results against actual application behavior. One of Selenium's key features is the support for executing one's tests on multiple browser platforms.

Brief History of The Selenium Project

Selenium first came to life in 2004 when Jason Huggins was testing an internal application at ThoughtWorks. Being a smart guy, he realized there were better uses of his time than manually stepping through the same tests with every change he made. He developed a Javascript library that could drive interactions with the page, allowing him to automatically rerun tests against multiple browsers. That library eventually became Selenium Core, which underlies all the functionality of Selenium Remote Control (RC) and Selenium IDE. Selenium RC was ground-breaking because no other product allowed you to control a browser from a language of your choice.

While Selenium was a tremendous tool, it wasn't without its drawbacks. Because of its Javascript based automation engine and the security limitations browsers apply to Javascript, different things became impossible to do. To make things worse, webapps became more and more powerful over time, using all sorts of special features new browsers provide and making these restrictions more and more painful.

In 2006 a plucky engineer at Google named Simon Stewart started work on a project he called WebDriver. Google had long been a heavy user of Selenium, but testers had to work around the limitations of the product. Simon wanted a testing tool that spoke directly to the browser using the 'native' method for the browser and operating system, thus avoiding the restrictions of a sandboxed Javascript environment. The WebDriver project began with the aim to solve the Selenium' pain-points.

Jump to 2008. The Beijing Olympics mark China's arrival as a global power, massive mortgage default in the United States triggers the worst international recession since the Great Depression, The Dark Knight is viewed by every human (twice), still reeling from the untimely loss of Heath Ledger. But the most important story of that year was the merging of Selenium and WebDriver. Selenium had massive community and commercial support, but WebDriver was clearly the tool of the future. The joining of the two tools provided a common set of features for all users and brought some of the brightest minds in test automation under one roof. Perhaps the best explanation for why WebDriver and Selenium are merging was detailed by Simon Stewart, the creator of WebDriver, in a joint email to the WebDriver and Selenium community on August 6, 2009.

"Why are the projects merging? Partly because WebDriver addresses some shortcomings in selenium (by being able to bypass the JS sandbox, for example. And we've got a gorgeous API), partly because selenium addresses some shortcomings in WebDriver (such as supporting a

broader range of browsers) and partly because the main selenium contributors and I felt that it was the best way to offer users the best possible framework."

Terminology:

Selenium Core (aka Core) is a set of JavaScript scripts that control the browser, imitating user activity. These scripts are injected into the web page and executed according to a list of actions written in a special HTML-table-based command language (aka Selense), thus simulating user activity.

Selenium RC (aka Selenium Remote Control or Selenium 1) receives Selenium Core commands via HTTP and executes them on a remote machine, proxying the web browser in order to avoid the "same host origin" restriction. This also allows writing the tests in other languages like C#, Python, Perl, PHP, Java and Ruby (via language bindings for Selenium Core).

Selenium-WebDriver (aka WebDriver or Selenium 2) is a successor of Selenium RC. It does the same job, but in a different way: instead of injecting a JavaScript code into the browser to simulate user actions, it uses the browser's native support for automation (different for each browser). Also, instead of a dictionary-based API (used in Selenium RC), it offers the more convenient object-oriented API.

Selenium Server allows using Selenium-WebDriver on a remote machine.

Selenium IDE is a Firefox add-on that records user activity and creates a test case based on it. It can also play the tests back and save them as a program in different languages.

Selenium-Grid allows you run the tests on different machines against different browsers in parallel; in other words, it enables distributed test execution.

Selenese is a special "language" represented by a set of Selenium commands that run your tests. A sequence of these commands is called a test script.

Selenium IDE

Introduces Selenium IDE and describes how to use it to build test scripts. using the Selenium Integrated Development Environment. If you are not experienced in programming, but still hoping to learn test automation this is where you should start and you'll find you can create quite a few automated tests with Selenium IDE. Also, if you are experienced in programming, this chapter may still interest you in that you can use Selenium IDE to do rapid prototyping of your tests. This section also demonstrates how your test script can be "exported" to a programming language for adding more advanced capabilities not supported by Selenium IDE.

Very good tutorial on how to get started with Selenium IDE:

http://www.guru99.com/first-selenium-test-script.html

http://www.seleniumhq.org/docs/01_introducing_selenium.jsp

API Testing

An API is an interface intended to be interpreted by a computer. Since the interface boundary has to be technology agnostic, the inputs and outputs are normally text-formatted messages. Text messages may be easy for a human to read in order to interpret the inputs and outputs, but in-depth knowledge of the internals of the application is required to sufficiently test it.

This is the difference between black-box testing and white-box testing. In black-box testing, the tester is only concerned with the functionality of the overall application, exercising only the inputs and checking the outputs.

In white-box testing, the tester is much more concerned with internal operations of the application, exercising the individual paths the data can travel through the application.

API testing normally includes only the white-box testing approach.

Another difficulty is that testers are used to working from a set of provided business requirements. However, business requirements normally describe use cases at a fairly high level, and do not go down to the level of individual application functions.

It is not unusual that business analysts are not even aware of low level detail and the amount of granularity that everything has to be broken down into, and are not aware that software is able to be tested at this level.

Typically, project managers are not concerned with assigning time specifically to developing a rich and detailed API, let alone testing it. Paradoxically, if everything is done correctly, the API is the only place where business rules are coded and enforced. The end-user GUI only displays results and any error messages provided by the API.

Sadly, because of the above problems, developers are often left with testing their own code, if there is any testing at this level at all. And due to time pressures, unit tests often only glance at the positive tests – the happy path – and more in depth testing is left until later in the project.

Of course, looking at only the positive paths through the application is not really testing, but only checking that the software fulfils business requirements under the most optimal conditions.

The GUI usually exposes only part of the overall functionality and by design testers are not able to go really deep into the application code, especially when it comes to testing all the negative test cases. This leaves the internals of the application possibly exposed to all sorts of breakage and attacks by end-users, accidental or malicious.

Lastly, there are situations where the API is the final product – a public API. A simple Internet search will reveal there are thousands of public APIs for anything from GPS and mapping solutions to locating a radio station that is playing your favourite song right now. However, these APIs expose functionality that the developer hopes or thinks other developers/ users will want. In this case, the tester's role needs also to include a user advocate role, to make sure the offering is complete – everything that any user may need is all there. This is in addition to the multitude of hats, the tester already wears: functional testing, security testing, load testing, etc.

SOAP vs REST

SOAP

SOAP – Simple Object Access Protocol – is probably the better known of the two models.

SOAP relies heavily on XML, and together with schemas, defines a very strongly typed messaging framework. Every operation the service provides is explicitly defined, along with the XML structure of the request and response for that operation. Each input parameter is similarly defined and bound to a type: for example an integer, a string, or some other complex object.

All of this is codified in the WSDL – Web Service Description (or Definition, in later versions) Language. The WSDL is often explained as a contract between the provider and the consumer of the service. In programming terms the WSDL can be thought of as a method signature for the web service.

Example: A sample message exchange looks like the following.

A request from the client:

```
POST http://www.stgregorioschurchdc.org/cgi/websvccal.cgi HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://www.stgregorioschurchdc.org/Calendar#easter_date"
Content-Length: 479
Host: www.stgregorioschurchdc.org
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
<?xml version="1.0"?>
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:cal="http://www.stgregorioschurchdc.org/Calendar">
<soapenv:Header/>
<soapenv:Body>
   <cal:easter_date soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <year xsi:type="xsd:short">2014</year>
</cal:easter_date>
</soapenv:Body>
</soapenv:Envelope>
```

The response from the service:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2013 21:09:44 GMT
Server: Apache/2.0.52 (Red Hat)
SOAPServer: SOAP::Lite/Perl/0.52
Content-Length: 566
Connection: close
Content-Type: text/xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
   <namesp1:easter_dateResponse
xmlns:namesp1="http://www.stgregorioschurchdc.org/Calendar">
<s-gensym3 xsi:type="xsd:string">2014/04/20</s-gensym3>
</namesp1:easter_dateResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

From this example we can see the message was sent over HTTP. SOAP is actually agnostic of the underlying transport protocol and can be sent over almost any protocol such as HTTP, SMTP, TCP, or JMS. As was already mentioned, the SOAP message itself must be XML-formatted. As is normal for any XML document, there must be one root element: the Envelope in this case. This contains two required elements: the Header and the Body. The rest of the elements in this message are described by the WSDL.The accompanying WSDL that defines the above service looks like this (the details are not important, but the entire document is shown here for completeness):

```
<?xml version="1.0"?>
<definitions xmlns:tns="http://www.stgregorioschurchdc.org/Calendar"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
name="Calendar" targetNamespace="http://www.stgregorioschurchdc.org/Calendar">
<message name="EasterDate">
   <part name="year" type="xsd:short"/>
</message>
<message name="EasterDateResponse">
   <part name="date" type="xsd:string"/>
</message>
<portType name="EasterDateSoapPort">
   <operation name="easter_date" parameterOrder="year">
     <input message="tns:EasterDate"/>
     <output message="tns:EasterDateResponse"/>
   </operation>
</portType>
<binding name="EasterDateSoapBinding" type="tns:EasterDateSoapPort">
   <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="easter_date">
     <soap:operation soapAction="http://www.stgregorioschurchdc.org/Calendar#easter_date"/>
     <input>
      <soap:body use="encoded" namespace="http://www.stgregorioschurchdc.org/Calendar" encodingStyle="http://schemas.xmlsoap.org/soap/
      encoding/"/>
     </input>
     <output>
       <soap:body use="encoded" namespace="http://www.stgregorioschurchdc.org/Calendar" encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/"/>
     </output>
   </operation>
</binding>
<service name="Calendar">
   <port name="EasterDateSoapPort" binding="tns:EasterDateSoapBinding">
     <soap:address location="http://www.stgregorioschurchdc.org/cgi/websvccal.cgi"/>
   </port>
</service>
</definitions>
```

Notice that all the parts of the message body are described in this document. Also note that, even though this document is intended to be primarily read by a computer, it is still relatively easy for a person with some programming knowledge to follow.

WSDL

The WSDL defines every aspect of the SOAP message. It is even able to define whether any element or attribute is allowed to appear multiple times, if it is required or optional, and can even dictate a specific order the elements must appear in.

It is a common misconception that the WSDL is a requirement.

It is a common misconception that the WSDL is a requirement for a SOAP service. SOAP was designed before the WSDL, and therefore the WSDL is optional. Although arguably, it is significantly harder to interface with a web service that does not have a WSDL.

On the other hand, if a developer is asked to interface with an existing SOAP web service, he only needs to be given the WSDL, and there are tools that do service discovery - generate method stubs with appropriate parameters in almost any language from that WSDL. Many test tools on the market work in the same way - a tester provides a URL to a WSDL, and the tools generate all the calls with sample parameters for all the available methods.

Analysis

While the WSDL may seem like a great thing at first – it is self documenting and contains almost the complete picture of everything that is required to integrate with a service – it can also become a burden. Remember, the WSDL is a contract between you (the provider of the service) and every single one of your customers (consumers of the service).

WSDL changes also means client changes.

If you want to make a change to your API, even something as small as adding an optional parameter, the WSDL must change. And WSDL changes also means client changes - all your consumers must recompile their client application against this new WSDL. This small change greatly increases the burden on the development teams (on both sides of the communication) as well as the test teams. For this reason, the WSDL is viewed as a version lock-in, and most providers are very resistant to updating their API.

Furthermore, while SOAP offers some interesting flexibility, such as the ability to be transmitted over any transport protocol, nobody has really taken advantage of most of these. Thanks to how the Internet evolved, everything that matters runs over HTTP. There are new advances, but most of these are being hampered by infrastructure routers refusing to route non-standard HTTP traffic. Just consider: how long has the world been trying to switch over to IPv6?

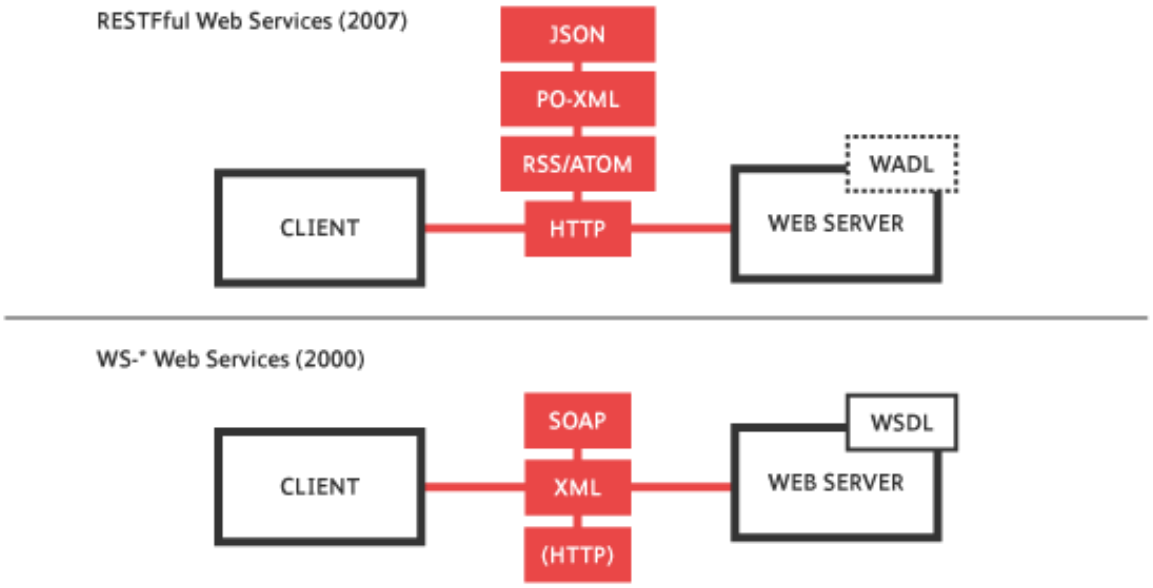
There is definitely a need for a more lightweight and flexible model [than SOAP].

Any situation where the size of the transmitted message does not matter, or where you control everything end-to-end, SOAP is almost always the better answer. This applies primarily to direct server to server communication, generally used for internal communication only within the confines of one company. However, there is a need for a world where almost every person on the planet has several low-memory, low-processing-power devices connected to multiple services at all times, there is definitely a need for a more lightweight and flexible model.

REST

REST – Representational State Transfer – is quickly becoming the preferred design model for public APIs. Some interesting statistics regarding the growth of REST in the public sphere are available at, for example, the Programmable Web.

REST is an architectural style, unlike SOAP which is a standardized protocol. REST makes use of existing and widely adopted technologies, specifically HTTP, and does not create any new standards. It can structure data into XML, YAML, or any other machine readable format, but usually JSON – JavaScript Object Notation – is preferred. As can be expected from JavaScript, the objects are not strongly typed. REST follows the object oriented programming paradigm of noun-verb. REST is very data-driven, compared to SOAP, which is strongly function-driven. In the REST paradigm, metadata is structured hierarchically and represented in the URI; this takes the place of the noun. The HTTP standard offers several verbs representing operations or actions you can perform on the data, most commonly: GET, POST, PUT, and DELETE.



For example, let's say we need three operations: a user login, logout, and retrieve the current user's account balance. While a SOAP service would implement each of these as separate operations (with the username and password passed as arguments to the login operation), REST would define a URI like http://sample.test/api/session. A POST action (with username and password passed in the body) to this URI would accomplish a user being logged into the session, and a DELETE action to the same URI would accomplish the session being terminated – effectively a logout. A GET action to say http://sample.test/api/session/balance would retrieve the current user's balance.

Example:

A sample message exchange could contain as little as this.

Request:

```
GET http://www.catechizeme.com/catechisms/catechism_for_young_children/daily_question.js HTTP/1.1
Accept-Encoding: gzip,deflate
Host: www.catechizeme.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Response:

```
HTTP/1.1 200 OK
Date: Fri, 22 Nov 2013 22:32:22 GMT
Server: Apache
X-Powered-By: Phusion Passenger (mod_rails/mod_rack) 3.0.17
ETag: "b8a7ef8b4b282a70d1b64ea5e79072df"
X-Runtime: 13
Cache-Control: private, max-age=0, must-revalidate
Content-Length: 209
Status: 200
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: js; charset=utf-8
{
   "link": "catechisms\/catechism_for_young_children\/questions\/36",
   "catechism": "Catechism for Young Children",
   "a": "Original sin.",
   "position": 36,
   "q": " What is that sinful nature which we inherit from Adam called?"
}
```

As is already expected this message was sent over HTTP, and used the GET verb. Further note that the URI, which also had to be included in the SOAP request, but there it had no meaning, here actually takes on a meaning. The body of the message is significantly smaller, in this example there actually isn't one.

A REST service also has a schema in what is called a WADL – Web Application Description Language. The WADL for the above call would look like this:

```
<?xml version="1.0"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
<doc xml:lang="en" title="http://www.catechizeme.com"/>
<resources base="http://www.catechizeme.com">
<resource path="catechisms/{CATECHISM_NAME}/daily_question.js" id="Daily_question.js">
<doc xml:lang="en" title="Daily_question.js"/>
<param xmlns:xs="http://www.w3.org/2001/XMLSchema" name="CATECHISM_NAME" style="template" type="string"/>
<method name="GET" id="Daily_question.js">
   <doc xml:lang="en" title="Daily_question.js"/>
   <request/>
   <response status="200">
     <representation mediaType="json" element="data"/>
     <representation mediaType="js; charset=utf-8" element="data"/>
   </response>
</method>
</resource>
</resources>
</application>
```

The WADL uses XML syntax to describe the metadata and the available actions. It can also be written to be as strict as the WSDL: defining types, optional parameters, etc.

WADL

The WADL does not have any mechanism to represent the data itself, which is what must be sent on the URI. This means that the WADL is able to document only about half of the information you need in order to interface with the service. Take for example the parameter CATECHISM_NAME in the above sample. The WADL only tells you where in the URI the parameter belongs, and that it should be a string. However, if you had to glean the valid values for yourself, it would probably take you quite a long time. Note that it is possible to add a schema to the WADL, so that you can define even complex variable types such as enumerations; however, this is even more rare than providing a WADL.

The WADL is completely optional.

Further the WADL is completely optional; in fact, it is quite rare that the WADL is supplied at all! Due to the nature of the service, in order to make any meaningful use of it, you will almost undoubtedly need additional documentation.

Analysis

Having a very small footprint and making use of the widely adopted HTTP standard makes REST a very attractive option for public APIs. Coupled together with JSON, which makes something like adding an optional parameter very simple, makes it very flexible and allows for frequent releases without impacting your consumers.

Arguably, the biggest drawback is the WADL – optional and lacking some necessary information. To address this deficiency, there are several frameworks available on the market that help document and produce RESTful APIs, such as Swagger, RAML, or JSON-home.

However, since there is no clear “standard” the domain flourishes with a multitude of different frameworks. This can make it hard to quickly grasp how the API works, if you prefer to work with clearly defined standards. On the other hand, if you thrive among disruptive technologies, then REST will probably be right up your alley. There are of course numerous discussions on best practices on the subject; one possible place to get you started is Martin Fowler's “Richardson Maturity Model” and Smartbear's "Understanding SOAP and REST Basic and Differences".

Advantages of API testing

Putting more effort into API testing leads to a much healthier final product. Ensuring that all data access (read and write) goes only through the API significantly simplifies security and compliance testing and thereby certification, since there is only one interface.

Ensuring that all the required business rules are being enforced at the API tier allows time for much more complete user-experience tests once the UI is released, and not having to concentrate on testing every single business rule and path through the application near the end of the project. Ensuring that the API offers complete functionality allows for easy future expansion of the application as new business needs arise.

Database Testing

SQL History

The origins of the SQL take us back to the 1970s, when in the IBM laboratories, new database software was created - System R. And to manage the data stored in System R, the SQL language was created. At first it was called SEQUEL, a name which is still used as an alternative pronunciation for SQL, but was later renamed to just SQL.

In 1979, a company called Relational Software, which later became Oracle, saw the commercial potential of SQL and released its own modified version, named Oracle V2.

Now into its third decade of existence, SQL offers great flexibility to users by supporting distributed databases, i.e. databases that can be run on several computer networks at a time. Certified by ANSI and ISO, SQL has become a database query language standard, lying in the basis of a variety of well established database applications on the Internet today. It serves both industry-level and academic needs and is used on both individual computers and corporate servers. With the progress in database technology SQL-based applications have become increasingly affordable for the regular user. This is due to the introduction of various open-source SQL database solutions such as MySQL, PostgreSQL, SQLite, Firebird, and many more.

SQL Standard

The SQL Standard has gone through a lot of changes during the years, which have added a great deal of new functionality to the standard, such as support for XML, triggers, regular expression matching, recursive queries, standardized sequences and much more. Due to SQL Standard's sheer volume, a lot of database solutions based on it, such as MySQL or PostgreSQL, do not implement the whole standard. In a lot of cases, the database behavior for file storage or indexes is not well defined and it's up to the vendors of the various SQL implementations to decide how the database will behave. This is the reason why, even though all SQL implementations have the same base, they are rarely compatible.

SQL Language elements

The SQL language is based on several elements. For the convenience of SQL developers all necessary language commands in the corresponding database management systems are usually executed through a specific SQL command-line interface (CLI).

- Clauses - the clauses are components of the statements and the queries

- Expressions - the expressions can produce scalar values or tables, which consist of columns and rows of data
- Predicates - they specify conditions, which are used to limit the effects of the statements and the queries, or to change the program flow
- Queries - a query will retrieve data, based on a given criteria
- Statements - with the statements one can control transactions, program flow, connections, sessions, or diagnostics. In database systems the SQL statements are used for sending queries from a client program to a server where the databases are stored. In response, the server processes the SQL statements and returns replies to the client program. This allows users to execute a wide range of amazingly fast data manipulation operations from simple data inputs to complicated queries.

SQL Queries

The SQL queries are the most common and essential SQL operations. Via an SQL query, one can search the database for the information needed. SQL queries are executed with the "SELECT" statement. An SQL query can be more specific, with the help of several clauses:

FROM - it indicates the table where the search will be made.

WHERE - it's used to define the rows, in which the search will be carried. All rows, for which the WHERE clause is not true, will be excluded.

ORDER BY - this is the only way to sort the results in SQL. Otherwise, they will be returned in a random order.

An SQL query example

```
SELECT * FROM
WHERE active
ORDER BY LastName, FirstName
```

SQL data control, definition and manipulation

SQL is a language designed to store data, but the data stored in an SQL database is not static. It can be modified at any time with the use of several very simple commands. The SQL syntax is pretty much self-explanatory, which makes it much easier to read and understand.

SQL data manipulation

Data manipulation is essential for SQL tables - it allows you to modify an already created table with new information, update the already existing values or delete them.

With the INSERT statement, you can add new rows to an already existing table. New rows can contain information from the start, or can be with a NULL value.

An example of an SQL INSERT

INSERT INTO phonebook(phone, firstname, lastname, address) VALUES('+1 123 456 7890', 'John', 'Doe', 'North America');

With the UPDATE statement, you can easily modify the already existing information in an SQL table.

An example of an SQL UPDATE

UPDATE phonebook SET address = 'North America', phone = '+1 123 456 7890' WHERE firstname = 'John' AND lastname = 'Doe';

With the DELETE statement you can remove unneeded rows from a table.

An example of an SQL DELETE

DELETE FROM phonebook WHERE firstname = 'John' AND lastname = 'Doe';

SQL data definition

Data definition allows the user to define new tables and elements.

CREATE - with the CREATE statement you can create a new table in an existing database.

An example of an SQL CREATE

CREATE TABLE phonebook(phone VARCHAR(32), firstname VARCHAR(32), lastname VARCHAR(32), address VARCHAR(64));

DROP - with the DROP statement in SQL you can delete tables, which you no longer need

An example of an SQL DROP

DROP TABLE phonebook;

TRUNCATE - with the TRUNCATE statement, you can delete all the content in the table, but keep the actual table intact and ready for further use

An example of an SQL TRUNCATE

TRUNCATE TABLE phonebook;

The ALTER statement permits the user to modify an existing object in various ways -- for example, by adding a column to an existing table.

ALTER TABLE phonebook RENAME TO contacts

SQL data control

SQL allows the user to define the access each of the table users can have to the actual table.

SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

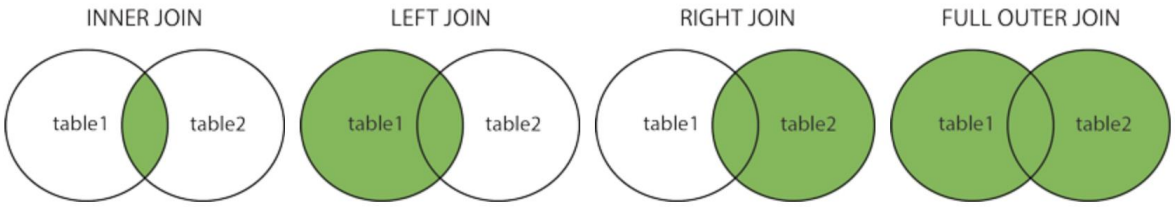
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate

FROM Orders

INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;

Here are the different types of the JOINs in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Return all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Return all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Return all records when there is a match in either left or right table



Learn more:

https://www.w3schools.com/sql/default.asp

https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

https://en.wikipedia.org/wiki/URL

https://www.upwork.com

https://www.soapui.org/

http://dret.net/netdret/docs/soa-rest-www2009/rest-ws.pdf

http://www.guru99.com/first-selenium-test-script.html