

**What Is Swagger?**

Swagger allows you to describe the structure of your APIs so that machines can read them.

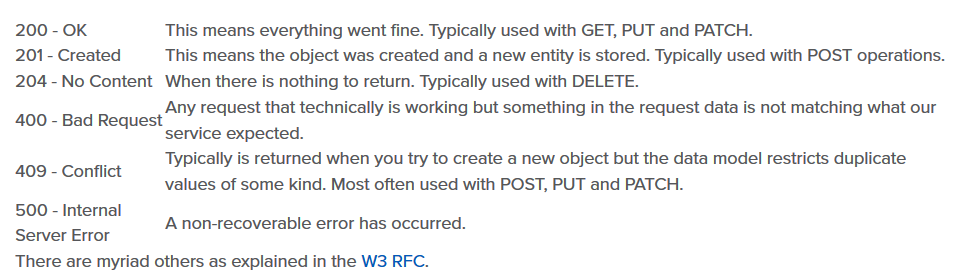
The ability of APIs to describe their own structure is the root of all awesomeness in Swagger. Why is it so great? Well, by reading your API’s structure, we can automatically build beautiful and interactive API documentation. We can also automatically generate client libraries for your API in many languages and explore other possibilities like automated testing.

Swagger does this by asking your API to return a YAML or JSON that contains a detailed description of your entire API. This file is essentially a resource listing of your API which adheres to [OpenAPI Specification](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md). The specification asks you to include information like:

* What are all the operations that your API supports?
* What are your API’s parameters and what does it return?
* Does your API need some authorization?
* And even fun things like terms, contact information and license to use the API.

You can write a Swagger spec for your API manually, or have it generated automatically from annotations in your source code. Check [swagger.io/open-source-integrations](https://swagger.io/open-source-integrations/) for a list of tools that let you generate Swagger from code.

**Response Codes**



**SOAP vs REST**

# **REST relies on** HATEOAS

**Hypermedia As The Engine Of Application State** (**HATEOAS**) is a constraint of the [REST application architecture](https://en.wikipedia.org/wiki/Representational_state_transfer) that distinguishes it from other network application architectures.

With HATEOAS, a client interacts with a network application that application servers provide dynamically entirely through [hypermedia](https://en.wikipedia.org/wiki/Hypermedia). A REST client needs no prior knowledge about how to interact with an application or server beyond a generic understanding of hypermedia.

I think these are the crucial points to understand what REST is about, and how it differs from SOAP:

* REST is protocol independent. It's not coupled to HTTP. Pretty much like you can follow an ftp link on a website, a REST application can use any protocol for which there is a standardized URI scheme.
* REST is not a mapping of CRUD to HTTP methods. Read [this](https://stackoverflow.com/questions/19843480/s3-rest-api-and-post-method/19844272#19844272) answer for a detailed explanation on that.
* REST is as standardized as the parts you're using. Security and authentication in HTTP are standardized, so that's what you use when doing REST over HTTP.
* REST is not REST without [hypermedia](https://stackoverflow.com/a/29586455/1202421) and [HATEOAS](http://en.wikipedia.org/wiki/HATEOAS). This means that a client only knows the entry point URI and the resources are supposed to return links the client should follow. Those fancy documentation generators that give URI patterns for everything you can do in a REST API miss the point completely. They are not only documenting something that's supposed to be following the standard, but when you do that, you're coupling the client to one particular moment in the evolution of the API, and any changes on the API have to be documented and applied, or it will break.
* REST is the architectural style of the web itself. When you enter Stack Overflow, you know what a User, a Question and an Answer are, you know the media types, and the website provides you with the links to them. A REST API has to do the same. If we designed the web the way people think REST should be done, instead of having a home page with links to Questions and Answers, we'd have a static documentation explaining that in order to view a question, you have to take the URI stackoverflow.com/questions/<id>, replace id with the Question.id and paste that on your browser. That's nonsense, but that's what many people think REST is.
* This last point can't be emphasized enough. If your clients are building URIs from templates in documentation and not getting links in the resource representations, that's not REST. Roy Fielding, the author of REST, made it clear on this blog post: [REST APIs must be hypertext-driven](http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

With the above in mind, you'll realize that while REST might not be restricted to XML, to do it correctly with any other format you'll have to design and standardize some format for your links. Hyperlinks are standard in XML, but not in JSON. There are draft standards for JSON, like [HAL](http://stateless.co/hal_specification.html).

Finally, REST isn't for everyone, and a proof of that is how most people solve their problems very well with the HTTP APIs they mistakenly called REST and never venture beyond that. REST is hard to do sometimes, especially in the beginning, but it pays over time with easier evolution on the server side, and client's resilience to changes. If you need something done quickly and easily, don't bother about getting REST right. It's probably not what you're looking for. If you need something that will have to stay online for years or even decades, then REST is for you.

# Which is more secure: SOAP OR REST?

3 Answers

[Madhu Sudhan R](https://www.quora.com/profile/Madhu-Sudhan-R-1)

[Madhu Sudhan R](https://www.quora.com/profile/Madhu-Sudhan-R-1), Test Analyst at Cerner Healthcare Solutions (2016-present)

[Answered Sep 2](https://www.quora.com/Which-is-more-secure-SOAP-OR-REST/answer/Madhu-Sudhan-R-1)

It all depends on business needs, neither SOAP nor REST is completely secure it all depends on how well the Authentication techniques are used.

when should you use REST vs SOAP?

**Encryption Vs Encoding**

Encoding is the process of transforming data so that it may be **transmitted without danger over a communication channel** or stored without danger on a storage medium. For instance, computer hardware does not manipulate text, it merely manipulates bytes, so a text encoding is a description of how text should be transformed into bytes. Similarly, HTTP does not allow all characters to be transmitted safely, so it may be necessary to encode data using **base64** (uses only letters, numbers and two safe characters).

When encoding or decoding, the emphasis is placed on everyone having the same algorithm, and that algorithm is usually well-documented, widely distributed and fairly easily implemented. **Anyone is eventually able to decode encoded data**.

Encryption, on the other hand, applies a transformation to a piece of data that can only be reversed with specific (and **secret**) knowledge of how to decrypt it. The emphasis is on making it hard for anyone but the intended recipient to read the original data. An encoding algorithm that is kept secret is a form of encryption, but quite vulnerable (it takes skill and time to devise any kind of encryption, and by definition you can't have someone else create such an encoding algorithm for you - or you would have to kill them). Instead, the most used encryption method uses secret keys : the algorithm is well-known, but the encryption and decryption process requires having the same key for both operations, and the key is then kept secret. **Decrypting encrypted data is only possible with the corresponding key**.

**Encoding:**

1. Purpose: The purpose of encoding is to transform data so that it can be properly (and safely) consumed by a different type of system.
2. Used for: For maintaining data usability i.e.,to ensure that it is able to be properly consumed.
3. Data Retrieval Mechanism: No key and can be easily reversed provided we know what algorithm was used in encoding.
4. Algorithms Used: ASCII, Unicode, URL Encoding, Base64
5. Example: Binary data being sent over email, or viewing special characters on a web page.

**Encryption:**

1. Purpose: The purpose of encryption is to transform data in order to keep it secret from others.
2. Used for: For maintaining data confidentiality i.e., to ensure the data cannot be consumed by anyone other than the intended recipient(s).
3. Data Retrieval Mechanism: Original data can be obtained if we know the key and encryption algorithm used.
4. Algorithms Used: AES, Blowfish, RSA
5. Example: Sending someone a secret letter that only they should be able to read, or securely sending a password over the Internet.

AES 128 Algorithm for Encryption and Decryption (javax.crypto.Cypher)

**import** java.security.**\***;

**import** java.security.spec.InvalidKeySpecException;

**import** javax.crypto.\*;

**import** sun.misc.**\***;

**public** **class** AESencrp {

**private** **static** **final** **String** ALGO = "AES";

**private** **static** **final** byte[] keyValue =

**new** byte[] { 'T', 'h', 'e', 'B', 'e', 's', 't',

'S', 'e', 'c', 'r','e', 't', 'K', 'e', 'y' };

*Or user String str = “key”;*

*byte[] byteVal = Str.getBytes(“UTF8”);*

**public** **static** **String** encrypt(**String** Data) **throws** **Exception** {

**Key** key = generateKey();

Cipher c = Cipher.getInstance(ALGO);

c.init(Cipher.ENCRYPT\_MODE, key);

byte[] encVal = c.doFinal(Data.getBytes());

**String** encryptedValue = **new** BASE64Encoder().encode(encVal);

**return** encryptedValue }

**public** **static** **String** decrypt(**String** encryptedData) **throws** **Exception** {

**Key** key = generateKey();

Cipher c = Cipher.getInstance(ALGO);

c.init(Cipher.DECRYPT\_MODE, key);

byte[] decordedValue = **new** BASE64Decoder().decodeBuffer(encryptedData);

byte[] decValue = c.doFinal(decordedValue);

**String** decryptedValue = **new** **String**(decValue);

**return** decryptedValue;

}

**private** **static** **Key** generateKey() **throws** **Exception** {

**Key** key = **new** SecretKeySpec(keyValue, ALGO);

**return** key;

}

}