RESTFUL Design Pattern

REST

**Representational state transfer** (**REST**) is the [software architectural style](https://en.wikipedia.org/wiki/Software_architecture_styles_and_patterns) of the [World Wide Web](https://en.wikipedia.org/wiki/World_Wide_Web).REST is coordinated set of constraints, applied to the design of components in a distributed [hypermedia](https://en.wikipedia.org/wiki/Hypermedia) system, can lead to a higher-performing and more maintainable [software architecture](https://en.wikipedia.org/wiki/Software_architecture).

To the extent that systems conform to the constraints of REST they can be called RESTful. RESTful systems typically not always, communicate over [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP) with the same [HTTP verbs](https://en.wikipedia.org/wiki/HTTP_verbs) (GET, POST, PUT, DELETE, etc.) that web browsers use to retrieve [web pages](https://en.wikipedia.org/wiki/Web_page) and to send data to remote servers.REST systems interface with external systems as [web resources](https://en.wikipedia.org/wiki/Web_resource) identified by [Uniform Resource Identifiers](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier) (URIs).

REST is the underlying architectural principle of the web. The amazing thing about the web is the fact that clients (browsers) and servers can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts. The key constraint is that the server and client must both agree on the *media* used, which in the case of the web is *HTML*.

An API that adheres to the principles of *REST* does not require the client to know anything about the structure of the API. The server needs to provide whatever information the client needs to interact with the service. An *HTML form* is an example of this: The server specifies the location of the resource, and the required fields. The browser doesn't know in advance where to submit the information, and it doesn't know in advance what information to submit. Both forms of information are entirely supplied by the server.(This principle is called *HATEOAS*.)

Architectural constraints

* The architectural properties of REST are realized by applying specific interaction constraints to components, connectors, and data elements. One can characterise applications conforming to the REST constraints described in this section as "RESTful".
* If a service violates any of the required constraints, it cannot be considered RESTful. Complying with these constraints, and thus conforming to the REST architectural style, enables any kind of distributed hypermedia system to have desirable [non-functional properties](https://en.wikipedia.org/wiki/Non-functional_requirement), such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.

REST constraints are:

### Client–server

A uniform interface separates clients from servers. This [separation of concerns](https://en.wikipedia.org/wiki/Separation_of_concerns) means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the [portability](https://en.wikipedia.org/wiki/Software_portability) of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more [scalable](https://en.wikipedia.org/wiki/Scalability). Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.

### Stateless

The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication.

The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.

### Cacheable

clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

### Layered system

A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.

### 

### 

### Uniform interface

The uniform interface constraint is fundamental to the design of any REST service.The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

Identification of resources

Individual resources are identified in requests, for example using [URIs](https://en.wikipedia.org/wiki/Uniform_resource_identifier) in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as [HTML](https://en.wikipedia.org/wiki/HTML), [XML](https://en.wikipedia.org/wiki/XML) or [JSON](https://en.wikipedia.org/wiki/JSON), none of which are the server's internal representation.

Manipulation of resources

When a client holds a representation of a resource, including any [metadata](https://en.wikipedia.org/wiki/Metadata) attached, it has enough information to modify or delete the resource.

Self-descriptive messages

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an [Internet media type](https://en.wikipedia.org/wiki/Internet_media_type)(previously known as a [MIME](https://en.wikipedia.org/wiki/MIME) type). Responses also explicitly indicate their cacheability.

Ex:

**Request**

GET / Accept: application/json+userdb

**Response**

200 OK Content-Type: application/json+userdb

{

"version": "1.0", "links": [

{ "href": "/user",

"rel": "list",

"method": "GET"

},

{

"href": "/user",

"rel": "create",

"method": "POST" } ]

}

REST Architectural Views

1. Processor View
2. Connector View
3. Data View

RESTFUL Design Patten in IOS

You will build an iOS client that consumes a Spring-based RESTful web service. Specifically, the client will consume the service created in [Building a RESTful Web Service](https://spring.io/guides/gs/rest-service/).

The iOS client will be accessed through the iOS Simulator, and will consume the service accepting requests at:

http://rest-service.guides.spring.io/greeting

The service will respond with a [JSON](https://spring.io/understanding/JSON) representation of a greeting:

{"id":1,"content":"Hello, World!"}

The iOS client will send the ID and content into a view.

## Create a ViewController

The Model-View-Controller design patten When you created the project, a RestViewController was also created with an empty implementation.

First, modify the header file to include two properties, and a method signature.

Implementation

Rest/RestViewController.h

#import <UIKit/UIKit.h>  
  
@interface RestViewController : UIViewController  
  
@property (nonatomic, strong) IBOutlet UILabel \*greetingId;  
@property (nonatomic, strong) IBOutlet UILabel \*greetingContent;  
  
- (IBAction)fetchGreeting;  
  
@end

The greetingId and greetingContent properties are UILabel types. These properties are declared with the type qualifier of IBOutlet. By declaring these as an IBOutlet, they can easily be connected to the view. Note how the fetchGreeting method is also declared with a type qualifier, in this case IBAction. Like the properties, this allows the method to be connected to a control in the view, for example a button.

Now update the fetchGreeting method body in the main class file.

Rest/RestViewController.m

#import "RestViewController.h"  
  
@interface RestViewController ()  
  
@end  
  
@implementation RestViewController  
  
- (IBAction)fetchGreeting;  
{  
 NSURL \*url = [NSURL URLWithString:@"http://rest-service.guides.spring.io/greeting"];  
 NSURLRequest \*request = [NSURLRequest requestWithURL:url];  
 [NSURLConnection sendAsynchronousRequest:request  
 queue:[NSOperationQueue mainQueue]  
 completionHandler:^(NSURLResponse \*response,  
 NSData \*data, NSError \*connectionError)  
 {  
 if (data.length > 0 && connectionError == nil)  
 {  
 NSDictionary \*greeting = [NSJSONSerialization JSONObjectWithData:data  
 options:0  
 error:NULL];  
 self.greetingId.text = [[greeting objectForKey:@"id"] stringValue];  
 self.greetingContent.text = [greeting objectForKey:@"content"];  
 }  
 }];  
}  
  
- (void)viewDidLoad  
{  
 [super viewDidLoad];  
 [self fetchGreeting];  
}  
  
- (void)didReceiveMemoryWarning  
{  
 [super didReceiveMemoryWarning];  
 // Dispose of any resources that can be recreated.  
}  
  
@end

The fetchGreeting method is where the HTTP request happens. It uses NSURLConnection to send an asynchronous request to the specified URL. This particular method makes use of construct called a "block". Blocks are similar to closures. In this case, the block is passed to the completionHandler method parameter, meaning that on completion of the HTTP request the code within the block is executed.

If data is received and there is not an error when the HTTP request completes,NSJSONSerialization is used to read the data into an NSDictionary. Once the data is available in a dictionary, the "id" and "content" values are retrieved and assigned to the two labels that are defined in the header.

Benefits of RESTFUL Design Pattern

Security

* Encrypt communications
* Make sure you authenticate and authorize users before processing
* Good coding habits to avoid direct attacks

Performance

Both raw performance and scalability will go to REST due to the request following simple HTTP protocols. Most SOAP stacks use SAX parsing (event based parsing) which greatly improves the scalability of SOAP stacks, but there is a measurable impact to the overhead. SOAP has the normal HTTP processing overhead in addition to the XML parsing overhead. REST just has the HTTP processing overhead.

Complexity

Complexity of the program is getting reduced beacuse of the RESTFUL Design Pattern.