* **RESTful Service**
* **RESTful Service**
  + - Problems with Monolith
    - Components of a RESTful Service (Resource, URL, Resource Representation)
    - REST constraints
    - Richardson Maturity Levels

**Problems with Monolith**

* The large monolithic code base
* Overloaded IDE
* Continuous deployment is difficult
* Scaling the application can be difficult
* Obstacle to scaling development
* Requires a long-term commitment to a technology stack

**Reference:**

http://microservices.io/patterns/monolithic.html

https://dzone.com/articles/splitting-a-monolith-application-into-services

https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

https://dzone.com/articles/monolithic-vs-microservice-architecture

https://www.tatvasoft.com/blog/the-difference-between-micro-services-and-web-services/

* **Monolithic solution has a number of benefits:**

**Simple to develop** - the goal of current development tools and IDEs is to support the development of monolithic applications

**Simple to deploy** - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime

**Simple to scale** - you can scale the application by running multiple copies of the application behind a load balancer

* However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant:
  + The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It’s a downwards spiral.
  + Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are.
  + Overloaded web container - the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.
  + Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven’t been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.
  + Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently
  + Obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.
  + Requires a long-term commitment to a technology stack - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development. With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let’s imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It’s possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.

**Components of a RESTful Service**

* Resource
* URL
* Resource Representation

References:

<https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>

* **Resources and Resource Identifiers**

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. “today’s weather in Los Angeles”), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author’s hypertext reference must fit within the definition of a resource.

A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

REST uses a resource identifier to identify the particular resource involved in an interaction between components. REST connectors provide a generic interface for accessing and manipulating the value set of a resource, regardless of how the membership function is defined or the type of software that is handling the request. The naming authority that assigned the resource identifier, making it possible to reference the resource, is responsible for maintaining the semantic validity of the mapping over time (i.e., ensuring that the membership function does not change).

* **URL**

A Uniform Resource Locator (URL) is a form of URI that has sufficient information embedded within it (access scheme and address usually) to resolve and locate the resource.

Site: <http://www.ietf.org/rfc/rfc1738.txt>

* **Representations**

REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A representation is a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.

**REST Constraints**

* Client–server architecture
* Statelessness
* Cacheability
* Layered system
* Code on demand (optional)
* Uniform interface
* Resource identification in requests
* Resource manipulation through representations
* Self-descriptive messages
* Hypermedia as the engine of application state (HATEOAS)

Reference:

<https://en.wikipedia.org/wiki/Representational_state_transfer>

* **Client–server architecture**

The principle behind the client–server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interface across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

* **Statelessness**

The client–server communication is 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.

* **Cacheability**

As on the World Wide Web, 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.

* **Code on demand (optional)**

Servers can temporarily extend or customize the functionality of a client by transferring executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript.

* **Uniform interface**

The uniform interface constraint is fundamental to the design of any REST service.[2] It simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:

* **Resource identification in requests**

Individual resources are identified in requests, for example using URIs 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, XML or JSON, none of which are the server's internal representation.

* **Resource manipulation through representations**

When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource.

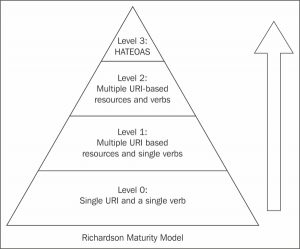
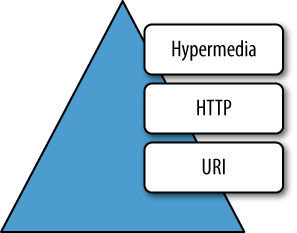
* **Self-descriptive messages**

Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by a media type.

* **Hypermedia as the engine of application state (HATEOAS)**

Having accessed an initial URI for the REST application—analogous to a human Web user accessing the home page of a website—a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs. As access proceeds, the server responds with text that includes hyperlinks to other actions that are currently available. There is no need for the client to be hard-coded with information regarding the structure or dynamics of the REST service.

**Richardson Maturity Levels**



References:

https://restfulapi.net/richardson-maturity-model/

https://dzone.com/articles/what-is-the-richardson-maturity-model-rmm

Richardson used three factors to decide the maturity of a service i.e. URI, HTTP Methods and HATEOAS (Hypermedia). The more a service employs these technologies – more mature it shall be considered.

In this analysis, Richardson described these maturity levels as below:

Level Zero

Level One

Level Two

Level Three

* **Level Zero**

Level zero of maturity do not make use of any of URI, HTTP Methods and HATEOAS capabilities.

These services have a single URI, and use a single HTTP method (typically POST). For example, most Web Services (WS-\*)-based services use a single URI to identify an endpoint, and HTTP POST to transfer SOAP-based payloads, effectively ignoring the rest of the HTTP verbs.

Similarily, XML-RPC based services which send data as Plain Old XML (POX). These are the most primitive way of building SOA applications with a single POST method and using XML to communicate between services.

* **Level One**

Level one of maturity makes use of URIs out of URI, HTTP Methods and HATEOAS.

These services employ many URIs but only a single HTTP verb – generally HTTP POST. They give each individual resource in their universe a URI. Every resource is separately identified by a unique URI – and that make them better than level zero.

* **Level Two**

Level two of maturity makes use of URIs and HTTP out of URI, HTTP Methods and HATEOAS.

Level two services host numerous URI-addressable resources. Such services support several of the HTTP verbs on each exposed resource – Create, Read, Update and Delete (CRUD) services. Here the state of resources, typically representing business entities, can be manipulated over the network.

Here service designer expect people to put some effort into mastering the APIs – generally by reading supplied documentation.

Level 2 is good use case of REST principles, which advocate using different verbs based on the HTTP request methods and the system can have multiple resources.

* Level Three

Level three of maturity makes use of all three i.e. URIs and HTTP and HATEOAS.

This is the most mature level of Richardson’s model which encourages easy discoverability and makes it easy for the responses to be self-explanatory by using HATEOAS.

The service leads consumers through a trail of resources, causing application state transitions as a result.

The Richardson Maturity Model (RMM) is a model developed by **Leonard** **Richardson**that helps organize your REST APIs into four levels, described below.

The main reason for this model to exist is to foster the evolution of the REST APIs of some system with a constant concern for governance, sense of organization with retro-compatibility, and constant improvements, using all the characteristics available from the REST architecture style.

* **Levels of RMM**
* **Level 0** (Swamp *of Pox*) - you should have a service already exposing a resource via its URI using only one HTTP verb (usually GET or POST). In this phase, you only use HTTP as a transport system (think of it as an RPC using HTTP).
* **Level 1** (*Resources*) - in this level you talk directly with your resources, correctly defined, using the appropriate division for it. Examples:

Resource '**hotel**':

Resource '**room**' (from Hotel):

* + services/hotel/1/rooms/all (or services/hotel/1/rooms) = *should bring details of all rooms of the hotel with ID #1.*
  + services/hotel/1/room/1 = *should bring details of the hotel room with ID #1 from hotel #01.*

*Here the details of a hotel could be its location, how many stars it has, how many rooms, etc.*

*Here details of the room could be the installation details, size, if it's occupied, etc.*

* **Level 2** (*HTTP Verbs*) - you have a clear understanding of the semantics of HTTP verbs and can now map your service operations using HTTP verbs accordingly, using [GET](https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html), [POST](https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html), [DELETE](https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html), [PUT](https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html) (most common verbs used in this Level), standardizing your access to your service's resources.
* **Level 3** (*Multimedia*) - you now can extract the maximum benefit from the HTTP protocol and services resources using Multimedia ([HATEOAS](https://en.wikipedia.org/wiki/HATEOAS)). This level the API should help with the discoverability of all resources associated with the payload requested, via links available within the payload itself.

**This model is heavily used with Microservices Architecture Style once you need to expose your services via on a standardized, easy-to-maintain, and organized API.**