REST Principles

The REST architectural style is intended to promote software longevity and independent evolution. RESTful APIs need to adhere to the constraints defined by Roy Fielding in his "Architectural Styles and the Design of Network-based Software Architectures" dissertation.

REST is constraints-driven rather than requirements-driven. When designing RESTful services the business domain should be mapped to architecture domain to achieve all the benefits. These constraints address the issues of network reliability, latency, bandwidth, security, network topology, administration, transport cost and heterogeneous network that are considerations in the development of distributed systems.

The following summarizes the REST constraints, for complete details refer to Fielding's dissertation.

1. Client-Server: Separation of concerns is the principle behind the client-server constraints as it allows the components to evolve independently, thus supporting the Internet-scale requirement.
2. Stateless: Communication must be stateless in nature, such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. This constraint induces the properties of visibility, reliability, and scalability.
3. Cache: Cache constraints have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions.
4. Layered System: The Layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, the overall system complexity is reduced/bounded and promotes substrate independence. This has the benefit of scalability of the system and different layers can be managed independently that further contributes to scalability.
5. Code on Demand (optional): REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.
6. Uniform Interface: The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components. By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. Uniform interface is defined by four constraints:
    1. Identification of resources: 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). REST uses resource identifier to identify a particular resource and in Web architecture URI is used as the resource identifier.
    2. Manipulation of resources through 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. Some examples of representation are HTML, XML, JSON etc. In Web architecture, REST uses HTTP methods to perform operations on resources.
    3. Self-descriptive messages: Messages between the components includes enough information on how to process the message. Message includes representation (HTML, JSON, etc), representation metadata (media-type, last-modified date, etc), resource metadata (source link, alternates, etc) and control data (if-modified-since, cache-control, etc)
    4. Hypermedia as the engine of application state: REST clients interacts with resources entirely through hypermedia provided dynamically by resources. The client needs no prior knowledge about how to interact with any particular resource beyond a generic understanding of hypermedia. Uniform interface constrain has the benefit of visibility enabling support of multiple platforms and systems can evolve independent of each other without affecting stability

REST Architectural Properties

Below are some of the key properties of RESTful architecture

1. Heterogeneity: REST constraints enable ability of an application or a system to interoperate with other application/system/services regardless of the language or platform
2. Scalability: REST constraints reduces the complexity between the components of a system. This enables an application to more efficiently manage requests and allowing it to scale horizontally.
3. Evolvability: The reduced complexity and dependency of system on one another allows for systems to evolve independent of one another.
4. Visibility: With uniform interface REST services are visible to clients, monitoring applications and gateways and being stateless there are no dependencies that consumers have to plan for
5. Reliability: Reliability can be planned for both on the client and service side. Both client and service can recover from failures by developing strategies that account for failure.
6. Efficiency: The constraint of cache allows for component such as proxy servers and cache to be included in the architecture allowing for better management of compute resources, bandwidth and load on the server.
7. Performance: The constraints of stateless eliminates the need for managing shared state on the server resulting in better utilization of available server resources and increased response time from the service.

8. Manageability: Constraint of client-server results in separation of concern resulting in managing evolution independently. The nature of interactions between components is predictable and consistent allowing for better integration of management tools.