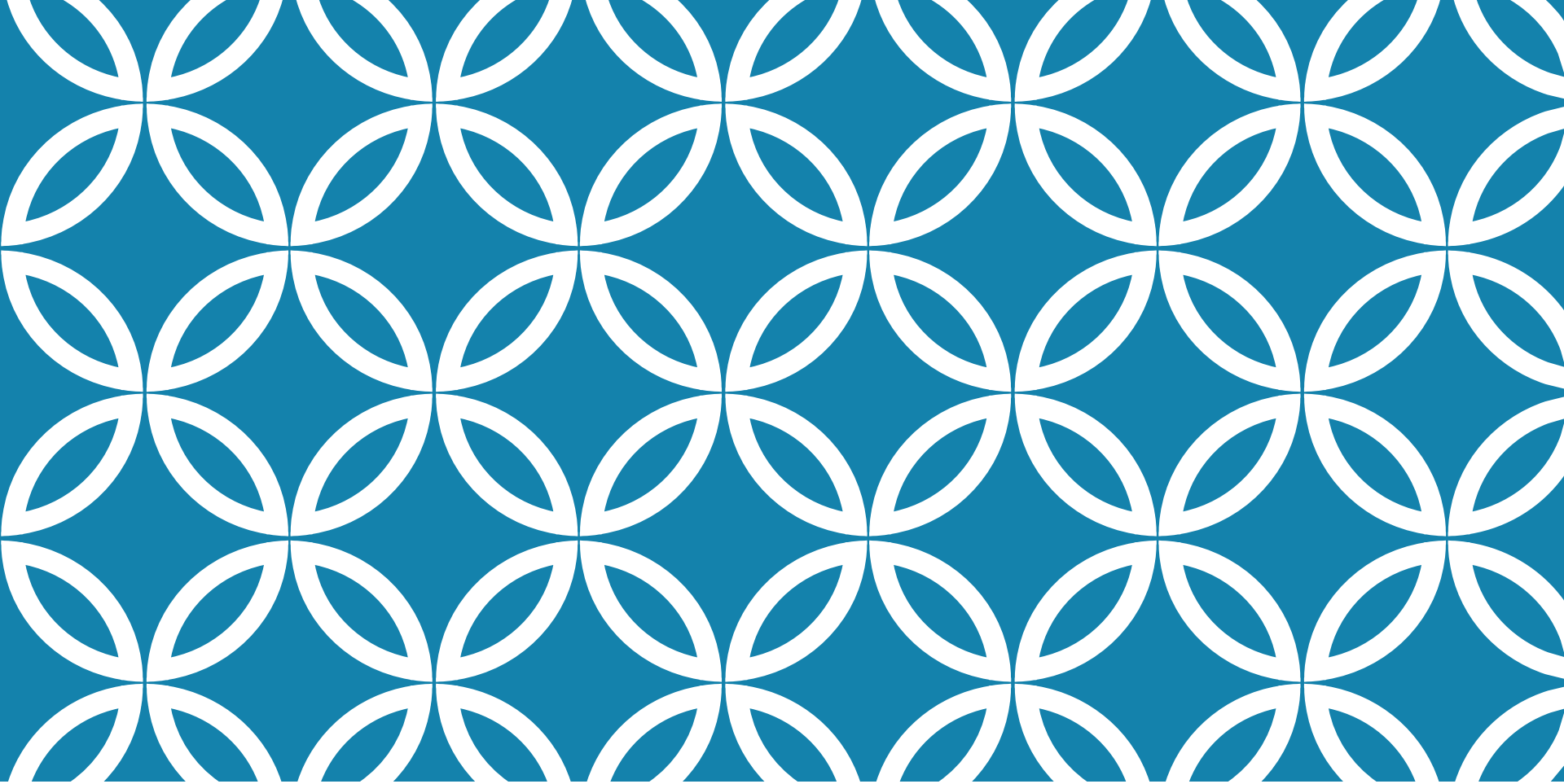

RESTFUL APPLICATIONS

October 2018

REST (**RE**PRESENTATIONAL **S**TATE **T**RANSFER)

REST is a set of design criteria and not the physical structure (architecture) of the system

- REST is not tied to the 'Web' i.e. doesn't depend on the mechanics of HTTP
- 'Web' applications are the most prevalent – hence RESTful architectures run off of it

UNDERSTANDING REST – RESOURCES

Anything that's important enough to be referenced as a thing in itself

Something that can be stored on a computer and represented as a stream of bits:

- A document
- A row in DB
- An output of executing an algorithm

URIS AND RESOURCES

- URI is an 'address' of a resource
- A resource must have *at least one* URI
- No URI → Not a resource
- URIs should be descriptive (human parseable) and have structure.

UNDERSTANDING REST - ADDRESSABILITY

- An application is addressable if it exposes *interesting aspects* of its data set as resources
- An addressable application exposes a URI for every piece of information it might conceivably serve (usually infinitely many ☺)
- Most important from end-user perspective
- Addressability allows one to *bookmark URIs* or embed them in presentations

REST PRINCIPLE #1

The key abstraction of information is a resource, named by a URI. Any information that can be named can be a resource

REST PRINCIPLE #2

All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

REST PRINCIPLE #3

The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components

UNDERSTANDING REST – UNIFORM INTERFACE

HTTP Provides 4 basic methods for CRUD (create, read, update, delete) operations:

- **GET**: Retrieve representation of resource
- **PUT**: Update/modify existing resource (or create a new resource)
- **POST**: Create a new resource
- **DELETE**: Delete an existing resource

Another 2 less commonly used methods:

- **HEAD**: Fetch meta-data of representation only (i.e. a metadata representation)
- **OPTIONS**: Check which HTTP methods a particular resource supports

HTTP REQUEST/RESPONSE

Method	Request Entity-Body/Representation	Response Entity-Body/Representation
GET	(Usually) Empty Representation/entity-body sent by client	Server returns representation of resource in HTTP Response
DELETE	(Usually) Empty Representation/entity-body sent by client	Server may return entity body with status message or nothing at all
PUT	(Usually) Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all
POST	Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all

REST PRINCIPLE #4

Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically

REST PRINCIPLE #5

Idempotent operations and representation metadata are encouraged in support of caching and representation reuse.

REST PRINCIPLE #6

The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server.

STEPS TO A RESTFUL ARCHITECTURE

Read the Requirements and turn them into resources ☺

1. Figure out the data set
2. Split the data set into resources

<u>For each kind of resource:</u>

3. Name resources with URIs
4. Expose a subset of uniform interface
5. Design representation(s) accepted from client (Form-data, JSON, XML to be sent to server)
6. Design representation(s) served to client (file-format, language and/or (which) status message to be sent)
7. Consider typical course of events: sunny-day scenarios
8. Consider alternative/error conditions: rainy-day scenarios

HTTP STATUS/RESPONSE CODES

HTTP is built in with a set of status codes for various types of scenarios:

- 2xx Success (*200 OK*, *201 Created*…)
- 3xx Redirection (*303 See other*)
- 4xx Client error (*404 Not Found*)
- 5xx Server error (*500 Internal Server Errror*)

Leverage existing status codes to handle sunny/rainy-day scenarios in your application!

BENEFITS OF RESTFUL DESIGN

- Simpler and intuitive design – easier navigability
- Server doesn't have to worry about client timeout
- Clients can easily survive a server restart (state controlled by client instead of server)
- Easy distribution – since requests are independent they can be handled by different servers
- Scalability: As simple as connecting more servers
- Stateless applications are easier to cache – applications can decide which response to cache without worrying about 'state' of a previous request
- Bookmark-able URIs/Application States
- HTTP is stateless by default – developing applications around it gets above benefits)