# Representational State Transfer

CMPU4023 - Enterprise Application Development

# What is REST?

- **Re**presentational **S**tate **T**ransfer (REST) is a way of abstracting and exposing service state and behaviour
- The big idea in REST is that service state and behaviour are modeled as resources
- The consumer interacts with one or more service resources using message passing API having a well-defined URI scheme over HTTP
- Service resources themselves can be backed by concrete state, such as a database table, or be synthetic in nature such as modelling behaviour of some service function (e.g. a service element is running or not)

# State Representation

- REST says nothing about the representation of state in the request and response messages
- Importantly, resource representation in REST is independent of actual internal service representation
- In practice, REST implementations use popular serialisation technologies such as JSON and XML for encoding state in messages

# Architectural Constraints

- **Client-Server**: Establishes the separation of concerns between the service provider and the service consumer. The provider offers one or more capabilities and listens for requests for those capabilities. The consumer is responsible for presenting the responses to the user and taking any corrective actions on foot of errors
- **Stateless**: The principle that no server-side state exists between any two REST requests from a consumer, i.e. requests are self-contained and standalone. Contrast this with a transactional style, for example

# Architectural Constraints (cont'd)

- **Cacheable**: Service responses can be explicitly labeled as cacheable or non-cacheable. This allows intermediating caching nodes to reuse previous responses to requests without having to go all the way back to the service. This is a key idea in making RESTful services scalable
- **Layered**: An arbitrary number of nodes can be placed between the ultimate service and service consumer. Their existence must be fully transparent so that they can be added and removed at will. This allows for the distribution and scalability of RESTful service solutions in practice

# Architectural Constraints (cont'd)

- **Uniform Contract**: This constraint states that all services and service consumers must share a single, overall technical interface. This is the primary constraint that really distinguishes REST from other architectures. In REST, this is applied using the verbs (commands) and media types of HTTP.

  The contract is usually free from specific business logic context so that it can be consumed by as wide a variety of client types as possible.

# Uniform Resource Identifier (URI)

- The URI is defined by an IETF standard which describes its overall form and what characters are legally permitted to be used
- A URI is made up from a URL (locator) and a URN (a unique name)
- The general form is as follows:

```
{scheme}://{authority}{path}?{query}
```

- Where:
  - Scheme is the transfer protocol (e.g. `http` or `https`)
  - Authority is the service address (e.g. `api.example.com`)
  - Path is the fully-qualified resource name including the URN (e.g. `/users/1234`)
  - Query is an optional set of qualifying parameters (e.g. `limit=100`)

# Example URI

- The following URI from the github API illustrates how URIs are formed

```
https://api.github.com/users?since=1000
```

- Here, the URI scheme is secured HTTP and the authority is **`api.github.com`**
- The path is **`/users`** which is the resource name (i.e. the users collection)
- The query parameter **`since`** specifies the user ID to start from, qualifying the API call
- A resource identifier is effectively globally unique though this says nothing about whether the resource itself is unique

# Resource Design

- In REST, a resource is just an independent representation of some service state or behaviour
- But it can be challenging to create a REST API that effectively models the service while also optimally serving the requirements of its consumers
- The problem is that the REST API designer has to try to anticipate how the API will be used and in what way
- Ideally the API development would take place with an active consumer helping to inform the design requirements throughout
- But this is not always possible and even when it is, it’s often no better than a snapshot in time

# Models

- A first-order approximation, in API design, would try to map resources to server-side models on a near one-to-one basis
- A ***model*** is the name given to some abstraction of internal (usually) persistent service state (e.g. a database table)
- For example, a products database table would be represented by a products model which would map to a RESTful products resource
- Following along those lines, you could imagine other tables and models which would represent orders, invoices, customer accounts and so on
- These in turn could be exposed as the corresponding resources in REST

# Resources Based on Models

- In the following we see examples of top-level resources and nested resources (often known as associations) based on models

| | |
|---|---|
| `/customers/123` | `Customer details for id 123` |
| `/customers/123/orders` | `The collection of orders for customer 123` |
| `/customers/123/invoices` | `The invoices for customer 123` |
| `/orders/456/products` | `The products associated with order 456` |
| `/products/789` | `Product details for id 789` |
| `/products?tag=electronic` | `List of products tagged as electronic` |

**ASIDE**: In the context of an RDBMS backing store, an association almost always indicates a table join is happening under the hood

# Views

- Whereas a model is the abstraction of the service state (e.g. a database table) a ***view*** is the representation of the model
- Views are the representation of a model using some serialisation format like XML or JSON
- Effectively when a resource is based on a model, the view also becomes the representation of the resource
- So-called model-view-controller (MVC) API builder frameworks take advantage of the model-and-view-as-resource convention to allow the automated generation of server-side API boilerplate code

# Aggregating Views

- In the simple model-to-resource-mapping approach, the API designer is taking a minimalist approach in the hope that most service consumers can always make use an API like this
- However, consumers will often want some kind of aggregated views of two or more related resources. In such cases are there two options

1. Create a new resource on the server side which provides the aggregation required (e.g. some nested structure comprised of multiple resources or resource fragments
2. Craft the aggregation on the client side by making whatever API calls are necessary to do so

- There are benefits and downsides to both approaches

# Aggregation Trade-offs

- Aggregating on the server has the benefit that it is available to all consumers
- The downside of a server-side approach is that it adds more complexity to the API implementation and forces it to specialise in some ways which may not even address the needs of all consumers
- Aggregating on the client has the advantage that exactly what is needed can be crafted for the consumer that needs
- The downside is usually that there is a performance penalty to having to make multiple API calls to fetch the necessary resources
- We will return to an alternative solution to this problem later in the module

# Summary

- REST is a design pattern for constructing message passing APIs over HTTP having a set of architectural constraints which distinguish it from other message passing approaches
- Service state and behaviour are abstracted as resources which are addressed using globally-unique resource identifiers
- REST APIs can be challenging to design as there is a trade-off between the cost and generality of implementation against the usefulness to any given consumer