For personal use only,
please do not
distribute on
non-UBC domains.

07b - REST APIs

CPSC310

Nick Bradley & Kyle Chin

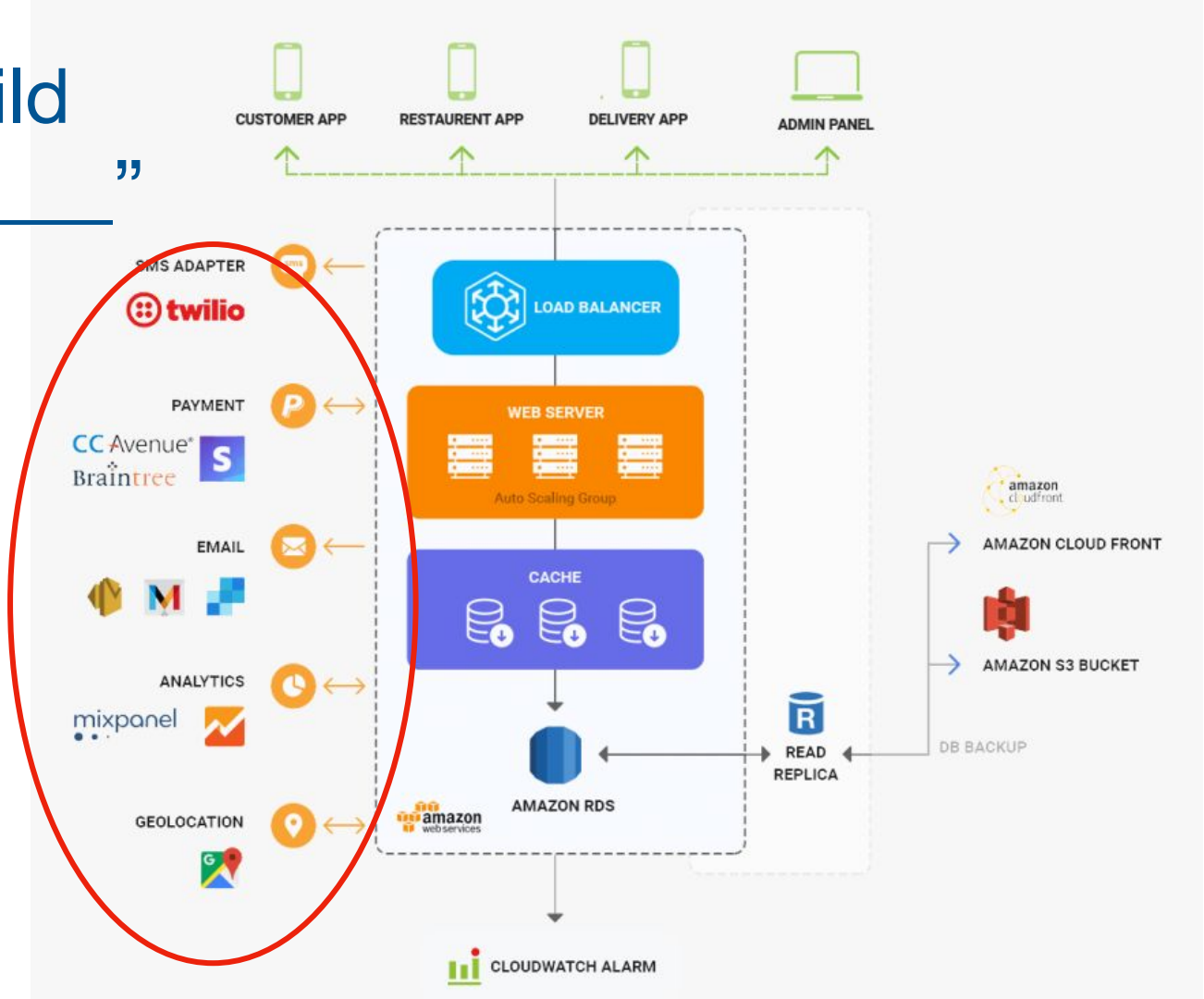
Learning goals

By the end of this lecture, you should be able to:

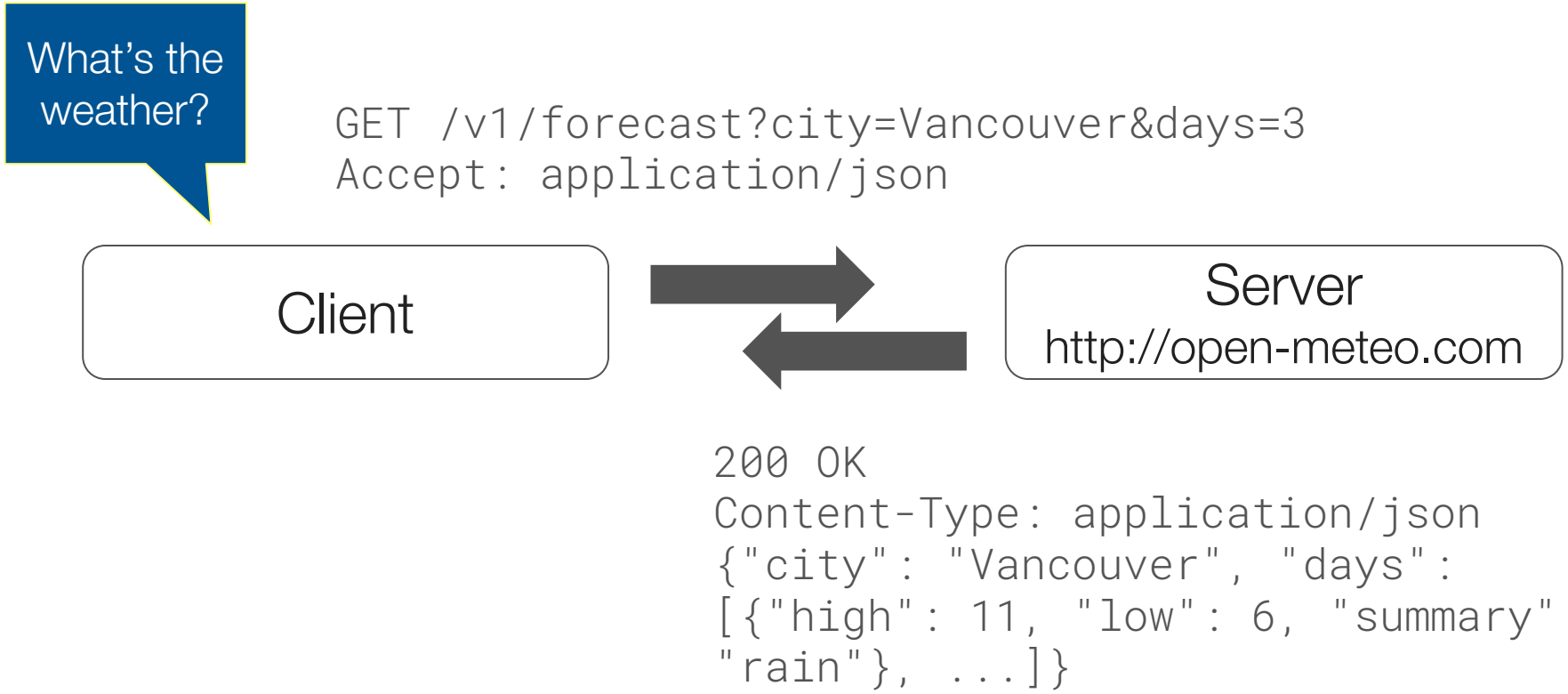
- Identify the parts of an HTTP request and response.
- Model requirements as resources acted on with GET, POST, PUT, and DELETE.
- Improve a design using the Uniform Interface, Idempotency, Statelessness, and Connectedness constraints.
- Recognize requirements that REST models poorly.

“I want to build the Uber of ____”

Software as a Service (SaaS) enabled Uber to build a full app without building everything from scratch!

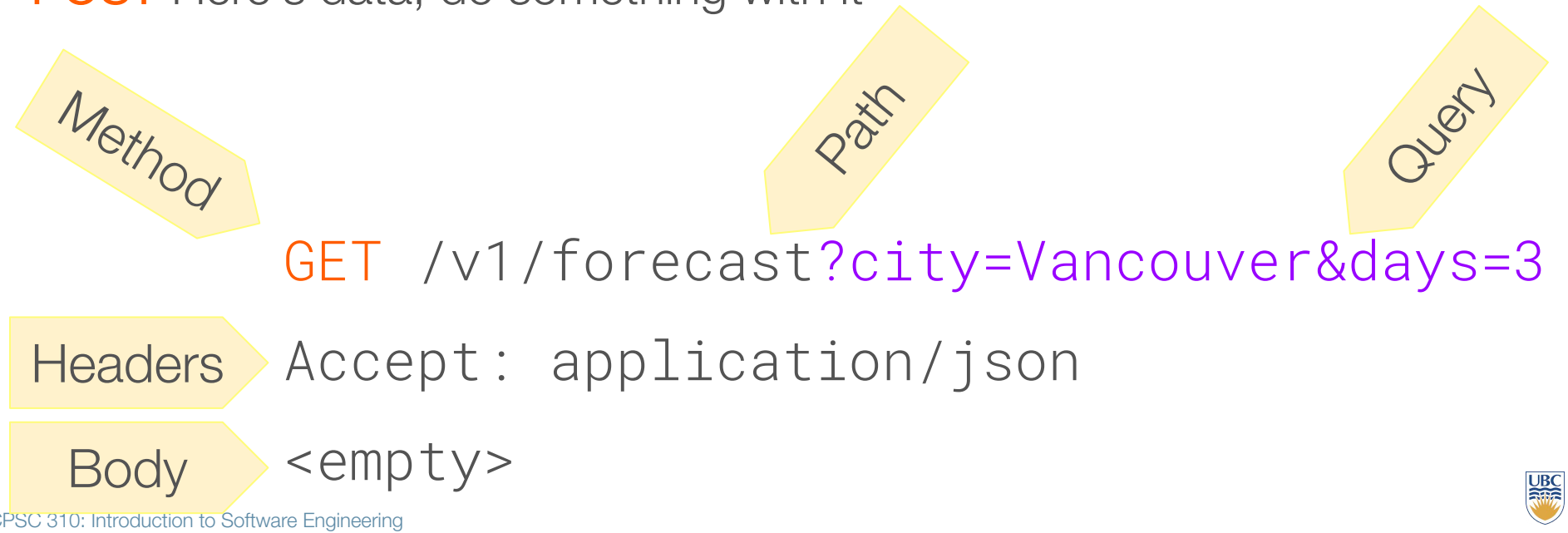


Example: Weather Service



Anatomy of an HTTP Request

GET Give me what's at the URL
PUT Store (replace) the given data at the URL
DELETE Remove what's at the path
POST Here's data, do something with it



Anatomy of an HTTP Response

2XX It worked (200 OK, 201 Created)
4XX Problem with client request (400 Bad Request, 404 Not Found)
5XX Server failed (500 Internal Server Error)

Status

```
200 OK
```

Headers

```
Content-Type: application/json
```

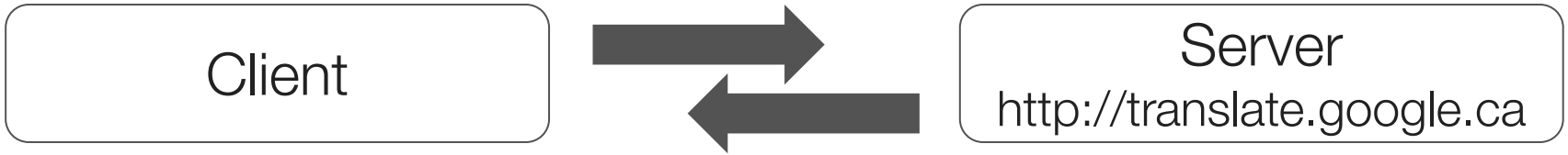
Body

```
{"city": "Vancouver", "days":
[{"high": 11, "low": 6}, ...]}
```

Another Example: Translation Service

What's this in French?

```
POST /translate
Accept: application/json
Content-Type: application/json
{"text": "Where is the bus loop?", "to": "fr"}
```




```
200 OK
Content-Type: application/text
Où se trouve la boucle des bus?
```

Scenario: Campus Bike Share



The university is launching a bike share with docking stations around campus. Bikes have ids printed on them (b1, b2, …). Riders use a phone app to find a station with bikes, enter the id of the bike, and return it at any station when they're done.

Modelling Resource Operations

Resource URI	Get (retrieve)	Put (update)	Post (create)	Delete (remove)
bikes/ (collection)	Returns the list of bikes (each one a URI)	Replaces the entire list. We don't want to do that, so nothing. **Send Error code**	Body: bike details Adds a bike to bikes/ Returns the URI for the bike.	Deletes the entire list. We don't want to do that, so nothing. **Send error code.**
bikes/{id} (a single bike)	Returns a particular bike and the sub-uris	Replaces the entire bike	Nothing to be done here. Error code or not impl.	Deletes the bike Get back a success code.
bikes/{id}/trips	Returns a list of trips for the bike	Do nothing.	Could be used to start a trip (risk: not idempotent)	Deletes all trips…so nothing.
bikes/{id}/trips/{id}	Returns a trip	Replaces entire trip (how we end)	Nothing	Deletes the trip.

RESTful Design Properties

Property	Definition	Problem it solved	Check
Uniform Interface	Clients act on resources using only standard methods, which mean the same thing in every API. Status codes report the outcome.	Standard HTTP libraries	Could a generic tool handle this request correctly without knowing your API?
Idempotency	Sending a request twice leaves the same state as sending it once. All but POST.	Retries at the dock	What happens if the client retries?
Statelessness	Each request carries everything needed to understand it. The server keeps the resources, not what the client has done.	/stations/more	Could a different server answer this request?
Connectedness	Responses link to related resources, so clients follow links instead of building URLs.	Changing the page size	Does the client ever build a URL itself?

A RESTful Bike Share Service

1. See which stations have bikes

   `GET /stations?hasBikes=true`

   → `200`, one page of stations, each with a `self` link, plus a `next` link

2. Take a bike (b1)

   `PUT /bikes/b1/trips/t-4f2a  {"rider": "alex"}`

   → `201`, the new trip, with links to its bike, rider and start station (a resend gets 200 and changes nothing)

3. Return it to a station (s2)

   `PUT /bikes/b1/trips/t-4f2a {..., "end": {"station": "s2"}}`

   → `200`, the ended trip

Where REST fits poorly

REST works well for many scenarios. But there are certain types or requirements that are hard to model:

- Live dock counts on the map, updated as bikes come and go.
- Unlocking the physical dock when the rider taps "unlock".
- A price quote for a ride between two stations.
- Reserve a bike and pay for it in one step

How would you model these? Why are they awkward?

Summary: REST APIs

- An HTTP request is a method, a URL, headers and a body; a response is a status, headers and a body.
- Model things, not actions: URLs as resources + methods as verbs. Actions, like a taking a trip, become a resource.
- Design for the network: use PUT for anything a client may resend, put the client's state in the request, and give clients links instead of making them build URLs.
- REST fits things you store and change. It's awkward for pushing updates, commands to hardware, and steps that must succeed together