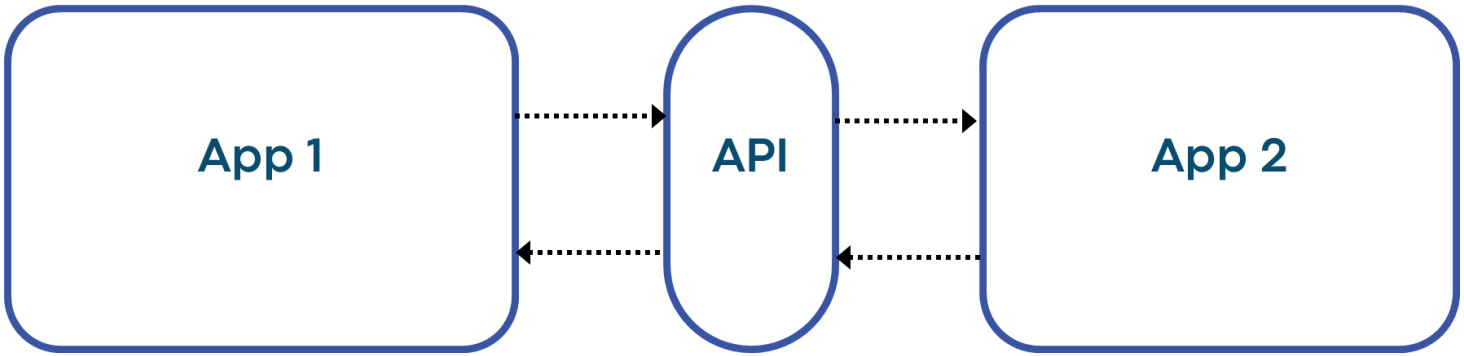
Web service design

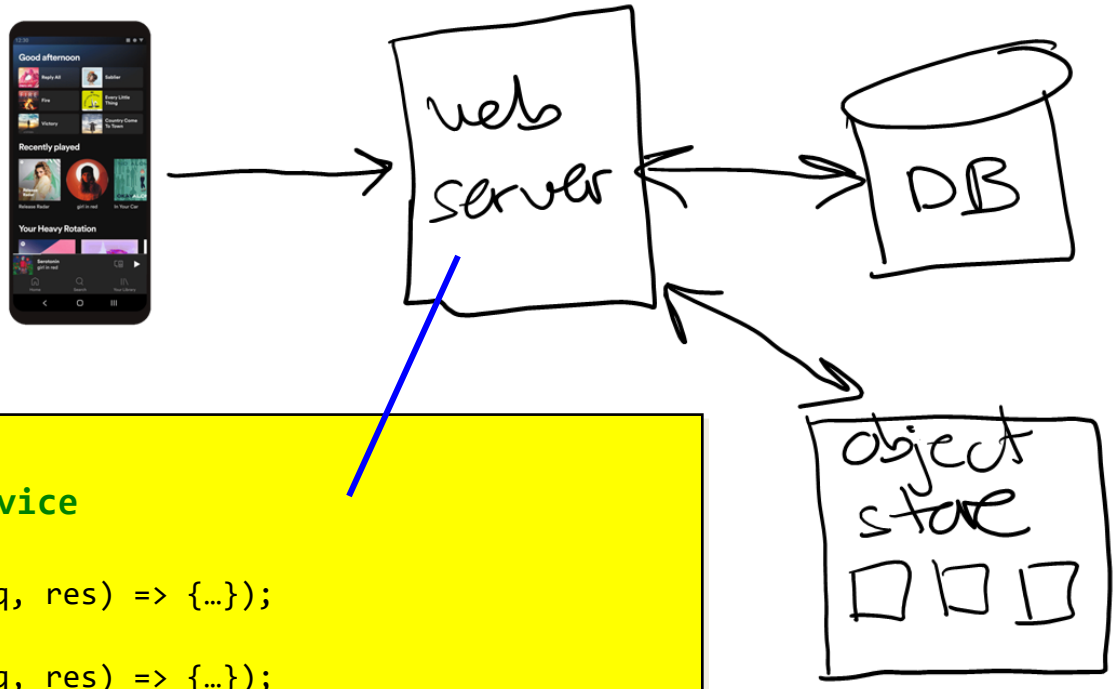
- Designing web services
- RESTful APIs
- Other design approaches
  - GraphQL

API

- **Programming interface between two applications / systems / servers**

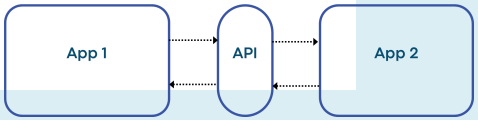


Example: Project 02




```
//
// PhotoApp web service
//
app.get('/stats', (req, res) => {…});

app.get('/users', (req, res) => {…});

app.get('/assets', (req, res) => {…});

app.get('/bucket', (req, res) => {…});

app.get('/image/:assetid', (req, res) => {…});

.
.
.
```

Design question



- *We are developing a social media application*
- *Our API needs to support a user "liking" an image. But a user can only like an image once, if they say they like the same image again and again, the result should still be just one "like".*
- *How would you design this functionality in the API?*

(A)

```
app.get('/like/image/:userid/:imageid', (req, res) => {
  ...
});
```

(B)

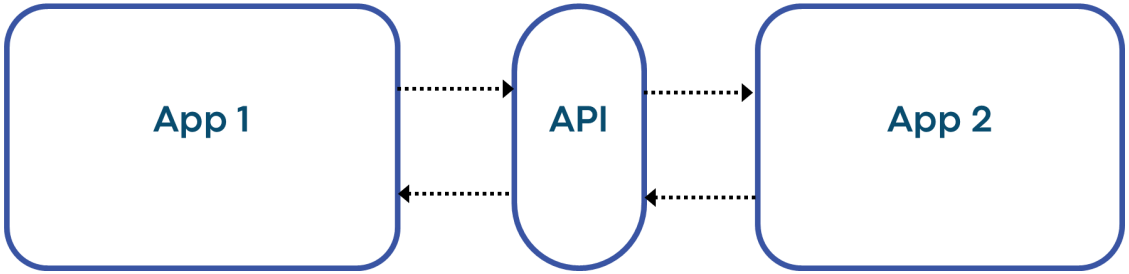
```
app.put('/like/image/:userid/:imageid', (req, res) => {
  ...
});
```

(C)

```
app.post('/like/image/:userid/:imageid', (req, res) => {
  ...
});
```

RESTful API

- **Many web service APIs are said to be "RESTful"**
- **This is an architectural style with design constraints:**



Constraints:

- Uniform Interface
- Stateless Operations
- Layered System
- Client-server based
- Cacheable
- Code on demand (optional)

that adheres to web server standards

Uniform means it adheres to web server standards like GET, POST, PUT, DELETE

Stateless Means, webservers do not remember anything. A request comes in, they perform some operation and forgot the transactions

Cahching mechanisms are deployed in places where servers are far away from the clients. To speed up the request response mechanism, caches are placed

RESTful APIs

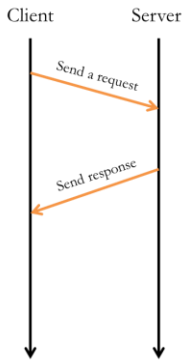
- **REST = REpresentational State Transfer** (PhD research)
- **RESTful APIs** are supposed to follow HTTP rules, and thus better integrate into web infrastructures
  - any function called via HTTP **get** should be *idempotent, i.e. compute the same result given the same system state*

    ==> *no side-effects*

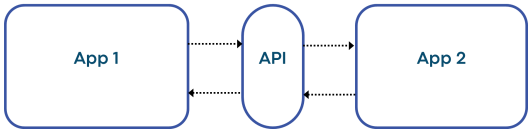
  - calling an idempotent function over and over again is supposed to be safe, and return the same value to support caching in servers closer to client

The benefit is suppose we cache a set of movies for a client who is far away.
Another client is also far way can take advantage of this cache and get timely results

Why would a request be repeated? If client doesn't get a timely response, they are allowed to repeat without causing side-effects…



Example



- **RESTful API for a social media app**

1. Uniform Interface
   - Endpoints refer to one resource

/users for all users

/users/:userid for one user

/posts

/posts/:postid

2. Stateless Operations
   - All info that my server needs to know should be in the request
   - Doesn't need info from a previous request

/posts/1/1000

/posts/1001/2000

Since web servers can not remember anything from previous request, everything we need, should be asked in a request.

Example: deleting a post?

- We are developing a social media application
- We are using HTTP DELETE to implement API function so users can delete their most recent post. Is design good or bad?

```
app.delete('/user/feed/posts/last/:userid', (req, res) => {
  .
  .
  .
});
```

This will repeatedly delete the latest post of the user if called again and again.
For eg
- Call 1 : Post n gets deleted
- Call 2 : Post n-1 is latest, it gets deleted
- Call 3 : Post n-2 is latest, its gets deleted

Ideally, the design should be, the API with same params should have the same effect i.e try to delete the same post

- This is a bad design because DELETE should be *idempotent*
  - With above design, repeated calls will keep deleting more and more posts
- Better design? Uniquely identify the post to delete, so repeated calls have no effect:

```
app.delete('/user/feed/posts/:userid/:postid', (req, res) => {
  .
  .
  .
});
```

HTTP requests and responses

Request methods:

- **GET**: to request data, if repeated should return the same data
- **PUT**: to put data on the server, if repeated should have same effect

This is basically an update query in the database if entry is already there

- **POST**: to post data to the server, if repeated treat as new data

This is an insert statement in the database. Does not care if entry is there or not

- **DELETE**: to remove data from the server, if repeated should have the same effect

Response codes

- **200 OK**: success
- **301 Moved Permanently**: redirects to another URL
- **400 Bad Request**: invalid client request
- **403 Forbidden**: lack permission
- **404 Not Found**: URL is bad
- **500 Internal Server Error**

… and many more

Design question #2

- Where to store the contents of a customer's **shopping cart**? Which do you think is the best option? This decision impacts the API design.

  A. Store cart in web server's memory?

  B. Store cart in the database?

  C. Store cart on the client?

One of these isn't RESTful (and doesn't scale)

One of these can lose the cart…

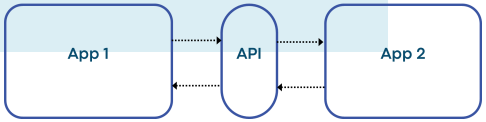
One of these involves extra network trip…

(A) --> This is not restful, as we are storing state of the system in web-server, which is meant to be stateless. It does not scale well, as the cart will be replicated across all servers, makes the client bound to one server

(C) —> It can lose the cart. As web browser is refreshed or closed, the cart is gone

(B) —> It is the best approach among them. It is stateful, it can be lost. The web-server also is stateless. One drawback is there is extra network trip

GraphQL API



- **GraphQL = graph query language** SQL for Web Servers
  - *a RESTful alternative…*
  - *Query-based vs. endpoint / function based*
  - *More powerful for clients / customers*
  - *Harder to build on the server*

Example

- **GitHub supports both RESTful and GraphQL APIs**
- **Example:**
  - *Given a username, get most-recent repos (last 3)*

REST

```
fetch('https://api.github.com/users/{username}');
fetch('https://api.github.com/users/{username}/repos');
```

Still need to parse data to get last 3 repos

https://api.github.com/users/joe-hummel

https://api.github.com/users/joe-hummel/repos

Advantage of GraphQL is ability to filter a lot of thigs with the query, without fetching bulk of results.

GraphQL

```
fetch('https://api.github.com/graphql', {
  method: 'POST', // using POST since GET can't have body
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    query: `{
      viewer {
        login,
        name,
        repositories(last: 3) {
          nodes {
            name
          }
        }
      }
    }`
  })
});
```

Similar to SQL

SELECT * FROM TABLE LIMIT 3

https://docs.github.com/en/graphql/overview/explorer

That's it, thank you!