Metodologie per la Programmazione per il Web - MF0437
Programmazione server-side con Express

Docente

Giancarlo **Ruffo**[giancarlo.ruffo@uniupo.it]

Informazioni, materiale e risorse su:

moodle [https://www.dir.uniupo.it/course/view.php?id=16455]

Slide adattate di versioni precedenti a cura dei

Proff. Luigi De Russis ed Alessio Bottrighi

Goal

- Getting started to implementing a web server
  - in JavaScript
  - for hosting static contents
  - for hosting dynamic APIs
  - supporting persistence in a Database
- Get to know REST and JSON

https://expressjs.com/
https://github.com/expressjs/express

Express

The Express Handbook, Flavio Copes
https://flaviocopes.com/page/express-handbook/

Web Frameworks in Node

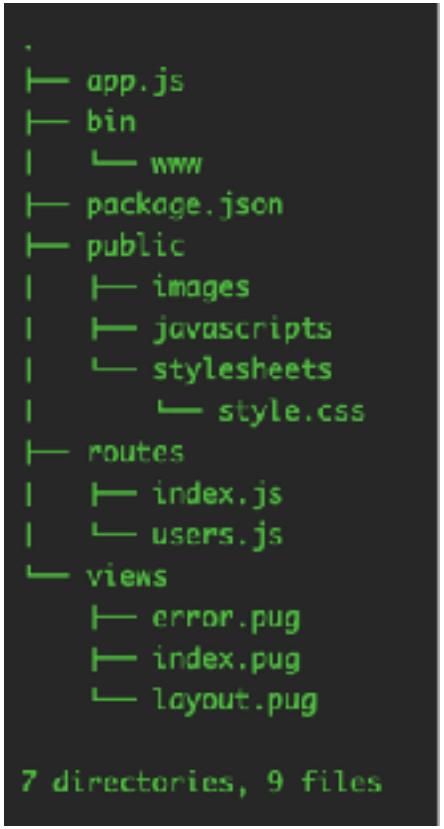
- Node already contains a 'http' module to activate a web server
  - low-level, non very friendly
- Several other frameworks were developed
- Express is among one of the most popular, and quite easy to use

```
npm init
npm install express
node index.js
```

http://nodeframework.com/

Express Projects Generator

- You can start a new Express project by using a **generator**
  - alternative way
  - pre-defined folder structure
  - includes view files for generating HTML pages
- `npx express-generator`
  - npx is a node command for running CLI tools and other executables
    - even if they are not installed
- after generating the project, you can install all its dependencies
  - `npm install`



https://blog.npmjs.org/post/162869356040/introducing-npx-an-npm-package-runner

First Steps with Express

- Calling `express()` creates an application object `app`
- `app.listen()` starts the server on the specified port (3000)
- Incoming HTTP request are routed to a callback according to
  - **path**, e.g., `'/'`
  - **method**, e.g., `get`
- Callback receives Request and Response objects (`req, res`)

```
// import package
const express = require('express') ;

// create application
const app = express() ;



// define routes and web pages
app.get('/', (req, res) =>
             res.send('Hello World!')) ;



// activate server

// last command in the page
app.listen(3000, () =>
      console.log('Server ready')) ;
```

Routing

- `app.method(path, handler);`
  - `app`: the express instance
  - `method`: an HTTP Request method (*`get`*, *`post`*, *`put`*, *`delete`*, …)
    - `app.all()` catches all request types
  - `path`: a path on the server
    - Matched with the path in the HTTP Request Message
  - `handler`: callback executed when the route is matched

```
app.get('/', (req, res) =>
res.send('Hello World!')) ;
```

Handler Callbacks

req (Request object)

```
function (req, res) { ... }
```

Property	Description
app	holds a reference to the Express app object
baseUrl	the base path on which the app responds
body	contains the data submitted in the request body (must be parsed and populated manually before you can access it)
cookies	contains the cookies sent by the request (needs the `cookie-parser` middleware)
hostname	the server hostname
ip	the server IP
method	the HTTP method used
params	the route named parameters
path	the URL path
protocol	the request protocol
query	an object containing all the query strings used in the request
secure	true if the request is secure (uses HTTPS)
signedCookies	contains the signed cookies sent by the request (needs the `cookie-parser` middleware)
xhr	true if the request is an XMLHttpRequest

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

https://expressjs.com/en/guide/routing.html

Generate a Response

- `res.send('something')` sets the response body and returns it to the browser
- `res.end()` sends an empty response
- `res.status()` sets the response status code
  - `res.status(200).send()`
  - `res.status(404).end()`
- `res.json()` sends an object by serializing it into JSON
  - `res.json({a:3, b:7})`
- `res.download()` prompts the user do download (not display) the resource

Redirects

- `res.redirect('/go-there')`

Extending Express with Middleware

- **Middleware**: a function that is called for every request
- `function(req, res, next)`
  - receives `(req, res)`, may process and modify them
  - calls `next()` to activate the next middleware function
- **Register** a middleware with
  - `app.use(callback)`
  - `app.use(path, callback)` // only requests in the specified path

Serving Static Requests

- Middleware: `express.static(root, [options])`
- All files under the root are served automatically
  - No need to register app.get handlers

```
app.use(express.static('public'));

Serves files from ./public as:
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

```
app.use('/static',
express.static('public'));

Serves files from ./public as:
http://localhost:3000/static/images/
kitten.jpg
http://localhost:3000/static/css/
style.css
http://localhost:3000/static/js/app.js
http://localhost:3000/static/images/
bg.png
http://localhost:3000/static/hello.html
```

https://expressjs.com/en/starter/static-files.html

Interpreting Request Parameters

Request method	Parameters	Values available in	Middleware requested

GET	URL-encoded
	`/login?user=alex&pass=stupidpwd`
		`req.query`
		`req.query.user`
		`req.query.pass`
			none

POST/PUT	FORM-encoded in the body	`req.body`
		`req.body.user`
		`req.body.pass`
			`express.urlencoded()`

POST/PUT	JSON stored in the body `{ "user": “alex", "pass": "stupidpwd" }`		`express.json()`

Paths

Path type	Example
Simple paths (String prefix)	`app.get('/abcd', (req, res, next)=> {`

Path Pattern (Regular expressions)	`app.get('/abc?d', (req, res, next)=> {`
	`app.get('/ab+cd', (req, res, next)=> {`
	`app.get('/ab\*cd', (req, res, next)=> {`
	`app.get('/a(bc)?d', (req, res, next)=> {`

JS regex object	`app.get(/\/abc|\/xyz/, (req, res, next)=> {`
Array (more than one path)	`app.get(['/abcd', '/xyza', /\/lmn|\/pqr/],` `(req, res, next)=> {`

https://expressjs.com/en/4x/api.html#path-examples

Parametric Paths

- A Path may contain one or more *parametric segments*:
  - Using the '`:id`' syntax
  - Free matching segments
  - Bound to an identifier
  - Available in `req.params`
- May specify a matching regexp
  - `/user/:userId(\d+)`

```
app.get('/users/:userId/
books/:bookId', (req, res) => {
  res.send(req.params)
});
```

Request URL: `http://localhost:3000/users/34/books/8989`

Results in:

```
req.params.userId == "34"
req.params.bookId == "8989"
```

https://expressjs.com/en/guide/routing.html#route-parameters

Modular Routes

- Use the `express.Router` class to create modular, mountable route handlers
  - a `Router` instance is a complete middleware and routing system, a sort of "mini-app"
- It exploits JavaScript modules

Modular Routes: Example

exams.js

```
const express = require('express');
const router = express.Router();


// define the home page route
router.get('/', function (req, res) {
   res.send('Exams home page');
});



// define the about route
router.get('/about', function (req, res) {
   res.send('About this app');
});



module.exports = router;
```

app.js

```
const express = require('express') ;
const exams = require('./exams');
const app = express();



// ...


app.use('/exams', exams);


//...


app.listen(3000);
```

Logging

- By default, express does not log the received requests
- For debugging purposes, it is useful to activate a logging middleware
- Example: `morgan`
  - https://github.com/expressjs/morgan (`npm install morgan`)
  - `const morgan = require('morgan');`
  - `app.use(morgan('tiny'));`

Validating Input

- https://express-validator.github.io/docs/
  - npm install express-validator
- Declarative validator for query parameters

```
app.post('/user', [   // additional (2nd) parameter in app.post to pre-process request
check('username').isEmail(), // username must be an email
check('password').isLength({ min: 5 }) // password must be at least 5 chars long
], (req, res) => {
const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(422).json({ errors: errors.array() });
  }
 . . . Process request
});
```

https://github.com/validatorjs/validator.js#validators

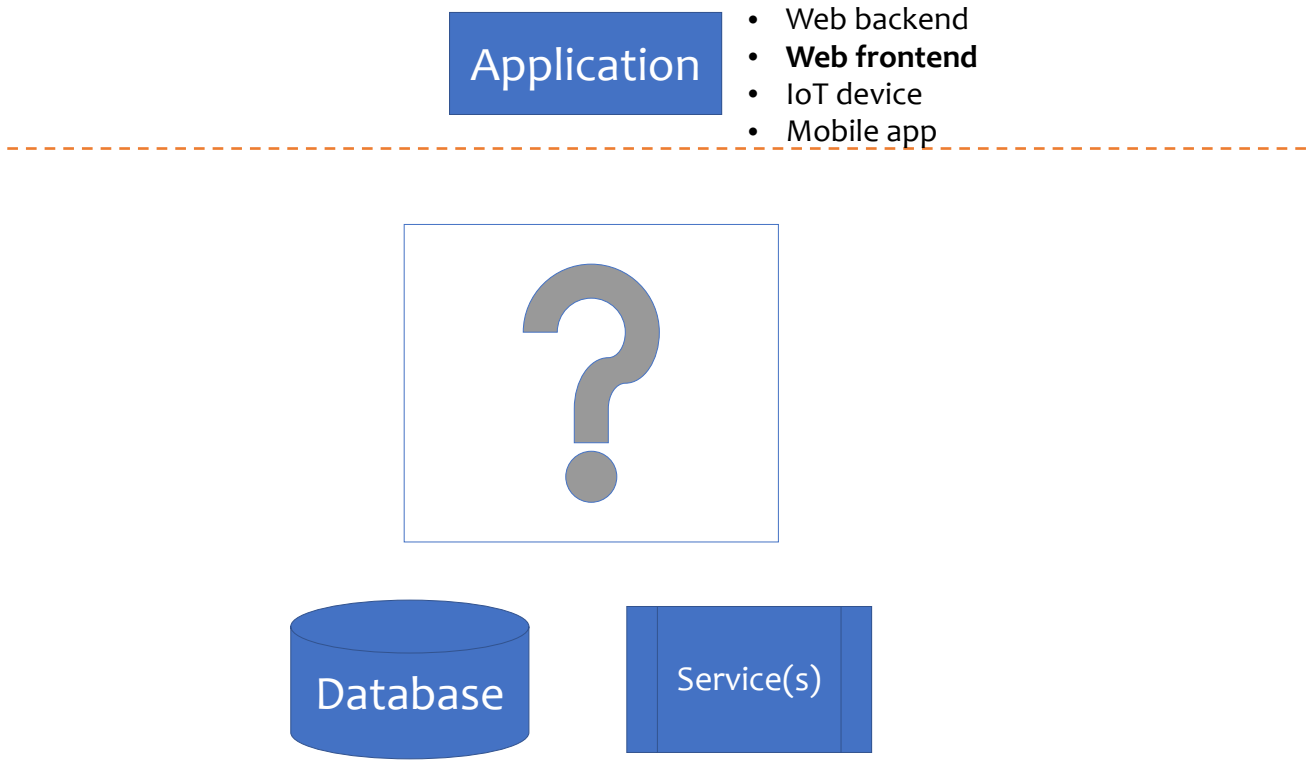
Other Middleware

Middleware module	Description	Replaces built-in function (Express 3)
body-parser	Parse HTTP request body. See also: body, co-body, and raw-body.	express.bodyParser
compression	Compress HTTP responses.	express.compress
connect-rid	Generate unique request ID.	NA
cookie-parser	Parse cookie header and populate req.cookies. See also cookies and keygrip.	express.cookieParser
cookie-session	Establish cookie-based sessions.	express.cookieSession
cors	Enable cross-origin resource sharing (CORS) with various options.	NA
csurf	Protect from CSRF exploits.	express.csrf
errorhandler	Development error-handling/debugging.	express.errorHandler
method-override	Override HTTP methods using header.	express.methodOverride
morgan	HTTP request logger.	express.logger
multer	Handle multi-part form data.	express.bodyParser
response-time	Record HTTP response time.	express.responseTime
serve-favicon	Serve a favicon.	express.favicon
serve-index	Serve directory listing for a given path.	express.directory
serve-static	Serve static files.	express.static
session	Establish server-based sessions (development only).	express.session
timeout	Set a timeout period for HTTP request processing.	express.timeout
vhost	Create virtual domains.	express.vhost

https://expressjs.com/en/resources/middleware.html

REST and JSON

Goal


Application
Web backend
Web frontend
IoT device
Mobile app
Database
Service(s)

REST

- **REpresentational State Transfer**

Roy T. Fielding

Senior Principal Scientist, Adobe
Co-founder, Apache HTTP Server Project
Director, The Apache Software Foundation
Ph.D., Information and Computer Science, UC Irvine

- @fielding; Blog: Untangled
- Email: fielding at (choose one of) gbiv.com, adobe.com, apache.org

- A *style of software architecture* for distributed systems
- Platform independent
  - you do not care if the server is Unix, the client is a Mac, or anything else
- Language independent
  - C# can talk to Java, etc.
- Standards based
  - runs on top of HTTP
- Can easily be used in the presence of firewalls

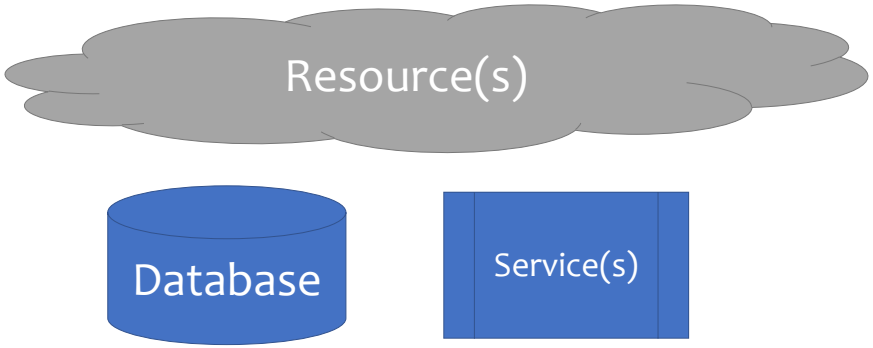
What is a Resource?

- A resource can be anything that has identity
  - a document or image
  - a service, e.g., "today's weather in New York"
  - a collection of other resources
  - non-networked objects (e.g., people)
- The resource is the **conceptual mapping** to an entity or set of entities
  - not necessarily the entity that corresponds to that mapping at any point in time!

REST Architecture

Application

- Web backend
- **Web frontend**
- IoT device
- Mobile app



Main Principles

- Resource: source of specific information
- Mapping: Resources ⇔ URIs
- Client and server exchange *representations* of the resource
  - the same resource *may* have different representations
  - e.g., XML, JSON, HTML, RDF, …

JSON - JavaScript Object Notation

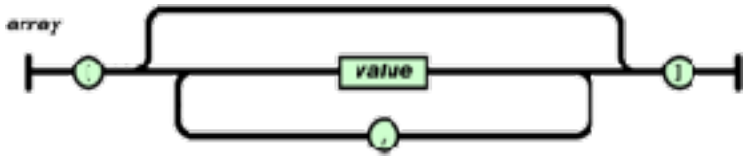
/ˈdʒeɪ·sən/

- Lightweight Data Interchange Format
  - Subset of JavaScript syntax for object literals
  - Easy for humans to read and write
  - Easy for machines to parse and generate
  - https://www.json.org/
  - ECMA 404 Standard: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
  - RFC 8259: https://tools.ietf.org/html/rfc8259
- Media type: `application/json`

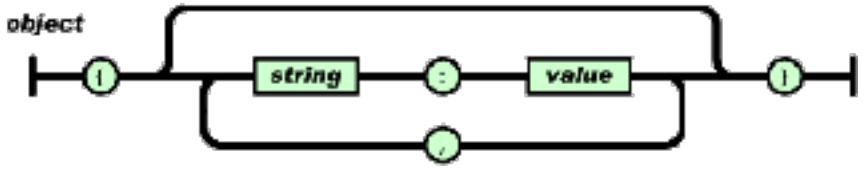
JSON Logical Structure

- Primitive types: string, number, true/false/null
  - Strings MUST use "double" quotes, not 'single'

- Composite type – Array: ordered lists of values



- Composite type – Objects: list of key-value pairs
  - Keys are strings (not identifiers)
  - MUST be "quoted"



JSON Example


```
{
    "firstName": "John",
    "lastName": "Smith",
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": 10021
    },
    "phoneNumbers": [
        "212 555-1234",
        "646 555-4567"
    ]
}
```

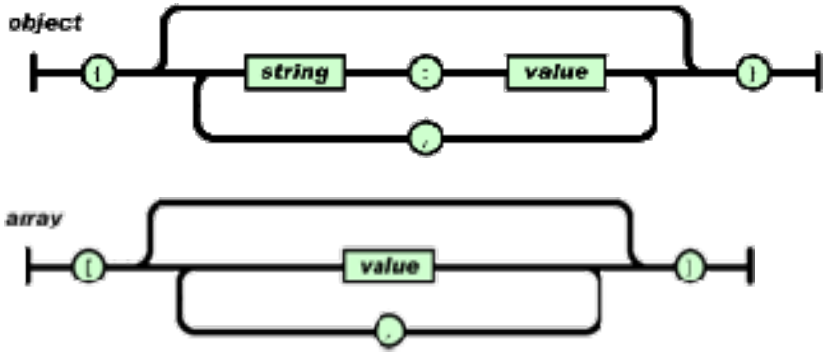
Name/Value Pairs

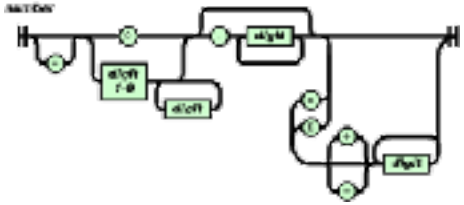
Child properties

String Array

Number data type

JSON Full Syntax



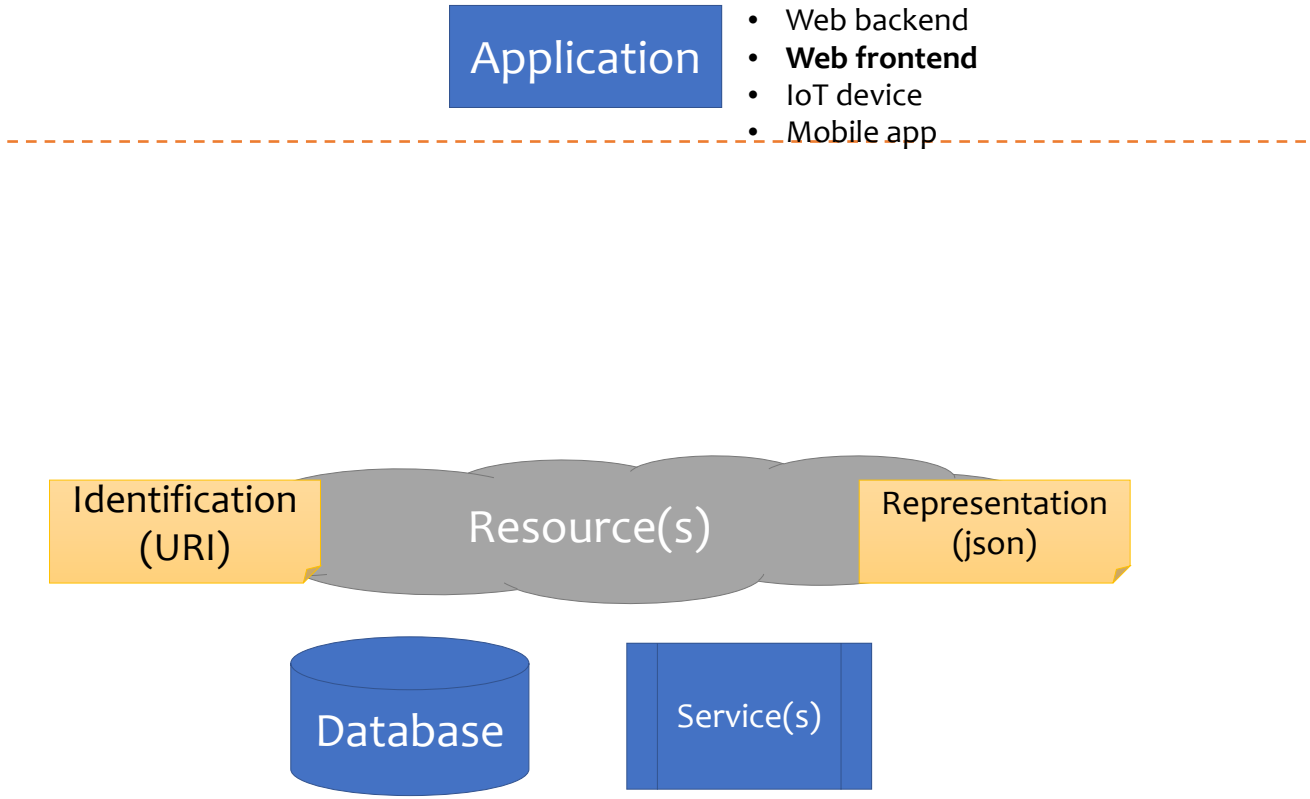


Using JSON in JavaScript

- `JSON.stringify` to convert objects into JSON
  - `const aString = JSON.stringify(myObj)`
  - Works recursively also on nested objects/arrays
  - Excludes function properties (methods) and undefined-valued properties
- `JSON.parse` to convert JSON back into an object
  - `const myObj = JSON.parse(aString)`
  - All created objects have the default `{}` Object prototype
    - Can fix with a *reviver* callback

https://javascript.info/json

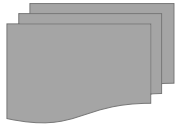
REST Architecture



Main Types of Resources

{REST}

- **Collection** resource
  - Represents a set (or list) of resources of the same type
  - Format: /resource
    - `http://api.uniupo.it/students`
    - `http://api.uniupo.it/courses`



- **Element** (Item, Simple) resource
  - Represents a single item, and its properties
  - Has some state and zero or more sub-resources
    - Sub-resources can be simple resources or collection resources
  - Format: /resource/identifier
    - `http://api.polito.it/students/s123456`
    - `http://api.polito.it/courses/S1729`

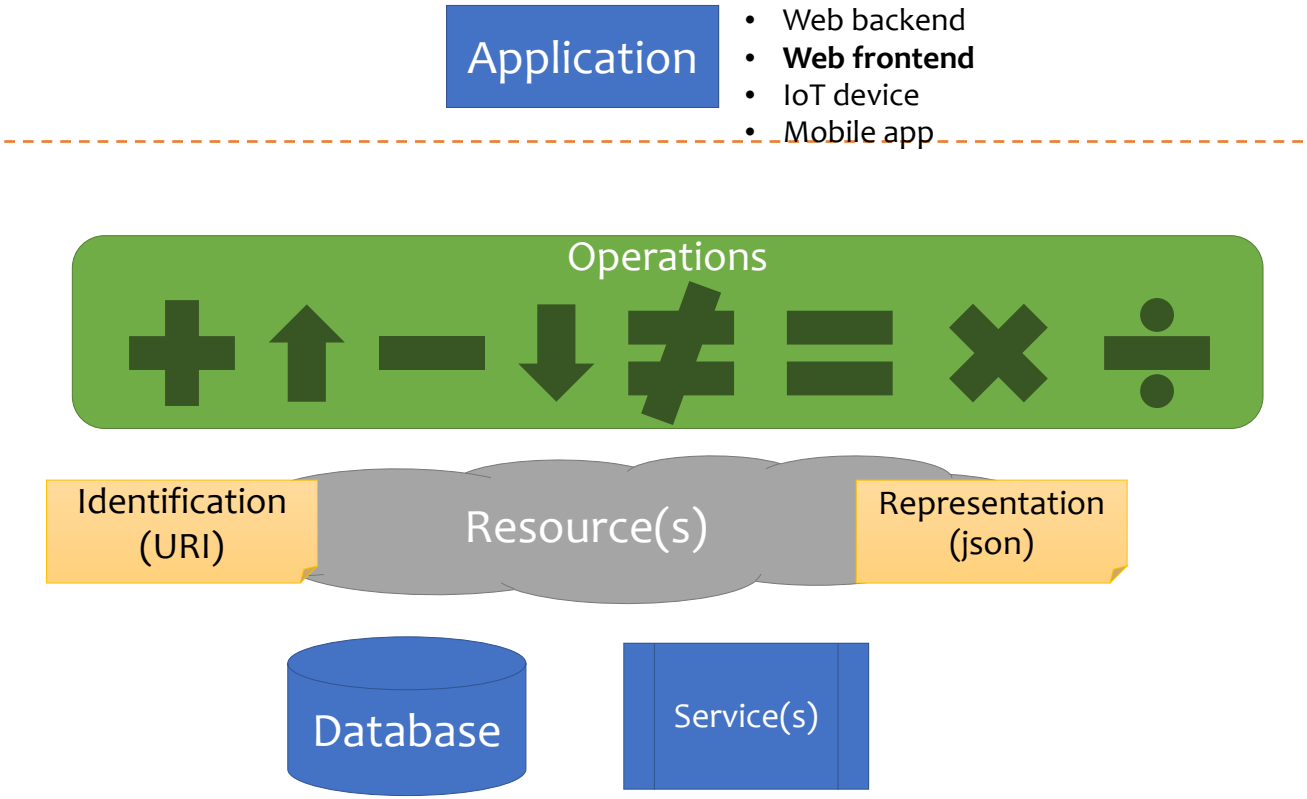
Best Practice

- Nouns (not verbs)
- Plural nouns
- Concrete names (not abstract)
  - /courses, not /items

Main Principles

- **Resources** support **Operations** (Actions)
  - Add
  - Delete
  - Update
  - Find
  - Search
  - …

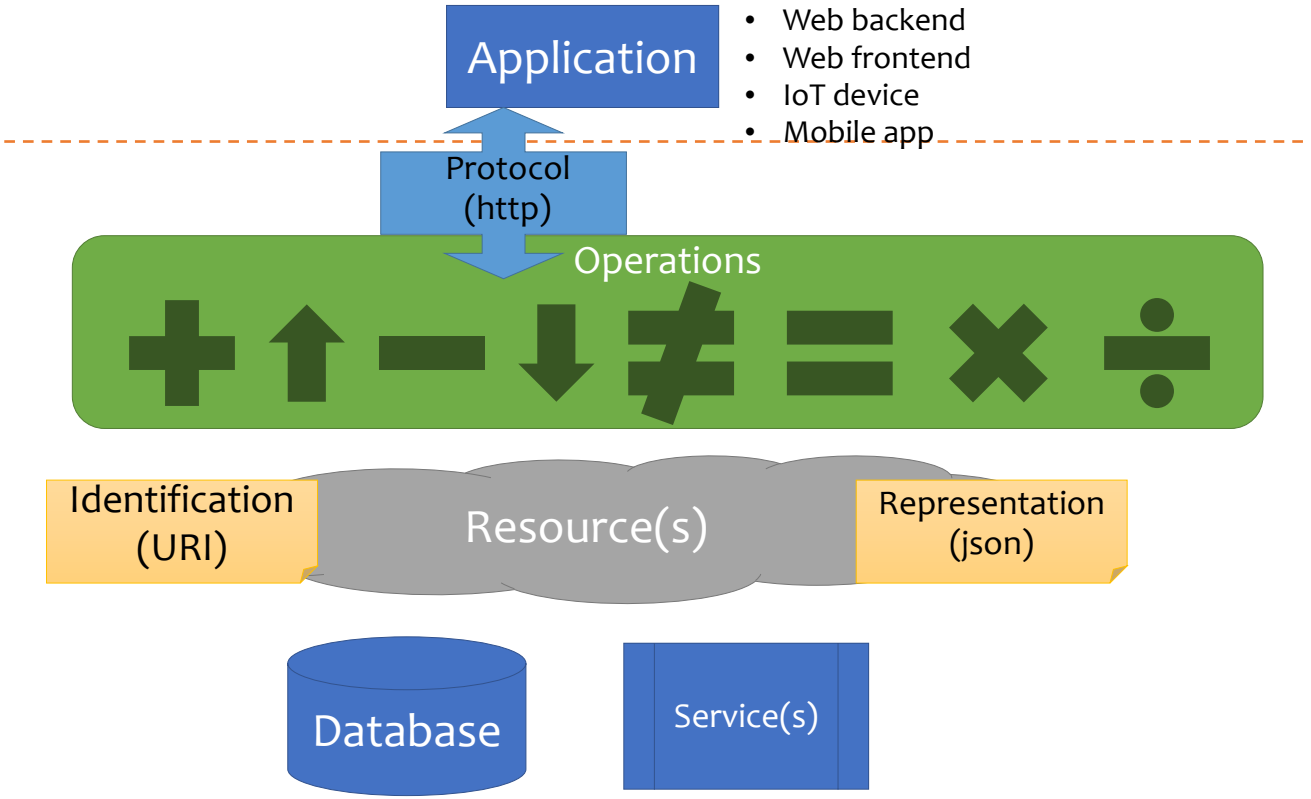
REST Architecture



Actions use HTTP Methods

- GET
  - Retrieve the representation of the resource (in the HTTP response body)
  - Collection: the list of items
  - Element: the properties of the element
- POST
  - Create a new resource (data in the HTTP request body)
  - Use a URI for a Collection
- PUT
  - Update an existing element (data in the HTTP request body)
  - Mainly for elements' properties
- DELETE

REST Architecture



Actions on Resources: Example

Resource	GET	POST	PUT	DELETE
/dogs	List dogs	Create a new dog	Bulk update dogs (avoid)	Delete all dogs (avoid)
/dogs/1234	Show info about the dog with id 1234	ERROR	If exists, update the info about dog #1234	Delete the dog #1234

Relationships

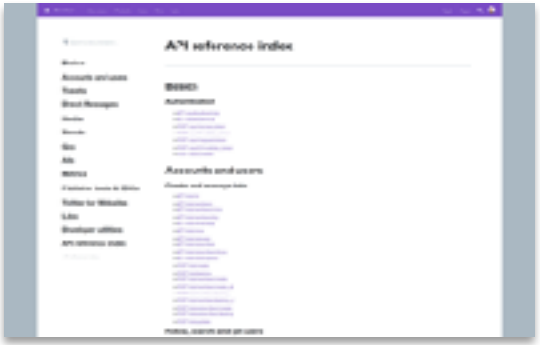
- A given Element may have a (1:1 or 1:N) relationship with other Element(s)
- Represent with: /resource/identifier/resource
  - `http://api.uniupo.it/students/s123456`/courses (list of courses followed by student `s123456`)
  - `http://api.uniupo.it/courses/S1729/students` (list of students enrolled in course `S1729`)

Representations

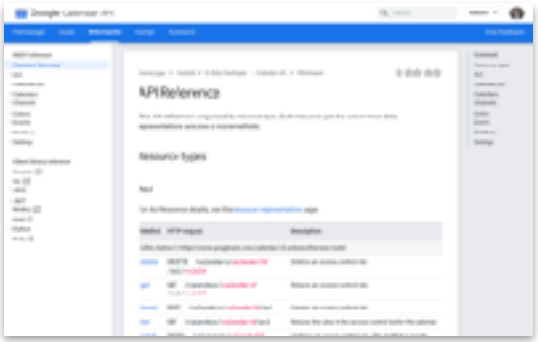
- Returned in GET, sent in PUT/POST
- Different formats are possible
- Mainly: XML, JSON
  - But also: SVG, JPEG, TXT, …
  - In POST: URL-encoding
- Format may be specified in
  - Request headers
    - **`Accept: application/json`**
  - URI extension
    - `http://api.uniupo.it/students/s123456.json`
  - Request parameter
    - `http://api.uniupo.it/students/s123456?format=json`

Real World Examples

https://developer.github.com/v3/



https://developer.twitter.com/en/docs/api-reference-index



https://developers.google.com/calendar/v3/reference/

https://developers.google.com/youtube/v3/docs

Complex resource search

- Use `?parameter=value` for more advanced resource filtering (or search)
  - E.g., `https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=twitterapi&count=2`

Errors

{REST}

- When errors or exceptions are encountered, use meaningful HTTP Status Codes
  - The Response Body may contain additional information (e.g., informational error messages)

```
{
   "developerMessage" : "Verbose, plain language
description of
the problem for the app developer with hints about
how to fix
it.",
   "userMessage":"Pass this message on to the app
user if
needed.",
   "errorCode" : 12345,
   "more info": "http://dev.teachdogrest.com/
errors/12345"
}
```

Guidelines (1/2)

URL Design	
Plural nouns for collections	/dogs
ID for entity	/dogs/1234
Associations	/owners/5678/dogs
HTTP Methods	POST GET PUT DELETE
Bias toward concrete names	/dogs (not animals)
Multiple formats in URL	/dogs.json /dogs.xml
Paginate with limit and offset	?limit=10&offset=0
Query params	?color=red&state=running
Partial selection	?fields=name,state
Use medial capitalization	"createdAt": 1320296464 myObject.createdAt;
Use verbs for non-resource requests	/convert?from=EUR&to=CNY&amount=100
Search	/search?q=happy%2Blabrador
DNS	api.foo.com developers.foo.com

Guidelines (2/2)

Versioning	
Include version in URL	/v1/dogs
Keep one previous version long enough for developers to migrate	/v1/dogs /v2/dogs

Errors	
Status Codes	200 201 304 400 401 403 404 500
Verbose messages	{"msg": "verbose, plain language hints"}

Client Considerations	
Client does not support HTTP status codes	?suppress_response_codes=true
Client does not support HTTP methods	GET /dogs?method=post GET /dogs GET /dogs?method=put GET /dogs?method=delete
Complement API with SDK and code libraries	1. JavaScript 2. … 3. …

REST API in Express

REST API Implementation

- REST API endpoints are just regular HTTP requests
- Request URL contain the Resource Identifiers (`/dogs/1234`)
  - extensive usage of parametric paths (`/dogs/:dogId`)
- Request/response Body contain the Resource Representation (in JSON)
  - `req.body` with `express.json()` middleware
  - `res.json()` to send the response
- Always validate input parameters
- Always validate input parameters
- Really, always validate input parameters

	Collections	Elements
GET	`app.get('/courses', (req, res) => { db.listCourses().then((courses) => { res.json(courses); }); });`	`app.get('/courses/:code', (req, res) => { // validation of req.params.code!! db.readCourse(req.params.code) .then((course)=>res.json(course)); });`

POST
PUT
		`app.use(express.json());`
		
		`app.post('/exams', (req, res) => { const exam = req.body; // validation of exam!! db.createExam(exam); });`

Data Persistence

Server-side Persistence

- The web server should normally store into a persistent database
- Node supports most databases
  - Cassandra, Couchbase, CouchDB, LevelDB, MySQL, MongoDB, Neo4j, Oracle, PostgreSQL, Redis, SQL Server, SQLite, Elasticsearch
- An easy solution for simple and small-volume applications is **SQLite** (in-process on-file relational database)

https://expressjs.com/en/guide/database-integration.html

SQLite

- Uses the 'sqlite' npm module
- Documentation: https://github.com/mapbox/node-sqlite3/wiki

```
npm install sqlite3
```

```
const sqlite = require('sqlite3');
const db = new sqlite.Database('exams.sqlite', //
DB filename
  (err) => { if (err) throw err; });


...
db.close();
```

SQLite: Queries

```
rows.forEach((row) => {
   console.log(row.name);
 });
```

- const sql = "SELECT…";
- `db.all(sql, [params], (err, rows) => { } )`
  - Executes sql and returns all the rows in the callback
  - If err is true, some error occurred. Otherwise, rows contains the result
  - Rows is an array. Each item contains the fields of the result
- `db.get(sql, [params], (err, row) => { } )`
  - Get only the first row of the result (e.g., when the result has 0 or 1 elements: primary key queries, aggregate functions, …)
- `db.each(sql, [params], (err, row) => { } )`
  - Executes the callback once per each result row (no need to store all of them)

https://www.sqlitetutorial.net/sqlite-nodejs/

Parametric Queries

- The SQL string may contain parameter placeholders: ?
- The placeholders are replaced by the values in the [params] array
  - In order: one param per each ?

```
const sql = 'SELECT * FROM course WHERE code=?';
db.get(sql, [code], (err, row) => {
```

- Always use parametric queries – *never* string+concatenation nor `template strings`

SQLite: Queries

- `db.run(sql, [params], (err) => { } )`
  - for statement that do not return a value
  - CREATE TABLE
  - INSERT
  - UPDATE
  - In the callback function
    - `this.changes` == number of affected rows
    - `this.lastID` == number of inserted row ID (for INSERT queries)

https://www.sqlitetutorial.net/sqlite-nodejs/

Alternatives

- Interacting with a database through a database driver (e.g., SQLite) is a *low-level* approach
  - it connects and works directly with a specific database and its SQL dialect
- *Middle-level* approach: query builders
  - libraries able to generate queries for a few different SQL dialects
  - example: knex, https://github.com/tgriesser/knex
- High-level approach: Object Relational Mapping (ORM)
  - libraries to map a record in a database to an object in our application
  - you define the structure of these objects, as well as their relationships, in code
  - example: sequelize, https://github.com/sequelize/sequelize

Alternatives: Example

SQLite

```
const sql =
"SELECT * FROM
change";


db.all(sql, (err,
rows) => {...} );
```

knex

```
knex('change').se
lect('*')

.then(rows =>
{ ... })

.catch(err =>
{... });
```

sequelize

```
Change.findAll()

.then(changes =>
{...});
```