

# Enterprise Programming 2

## Lesson 03:

## Charset, PATCH and GUI

Bogdan Marculescu

# Goals

- Understand how strings are represented in bytes when sent over the network
- Understand why all of a sudden your text is full of ? symbols or weird letters like Ã
- Understand the HTTP method PATCH, and how it is different from PUT
  - this is rather tricky, and a LOT of APIs out there do it wrong
- Learn how to integrate SPA library like React in SpringBoot

# Charsets

# Charsets

- Each character in a string text is mapped to a bitstring representation, which can be seen as a number
- But there are many types of mappings, called *Charsets*
- Tradeoff: how many characters we can represent vs the size in bits of the representation

# ASCII Codes

- **American Standard Code for Information Interchange (ASCII)**
- Mapping for 128 characters commonly used in English
  - Eg, a-z, A-Z, 0-9, ?, !, #, %, ...
- As  $128 = 2^7$ , we just need 7 bits, so can be stored in 1 byte
- Problem: how to represent special characters like the Norwegian øæåØÆÅ, or Japanese 私はアンドレアです???

# Unicode

- Standard for encoding of characters
- Representing up to 1,114,112 possible characters
- Currently mapping around 136,755 characters used in most languages around the world
  - unfortunately, in 2001 the Unicode Technical Committee rejected the proposal of adding *Klingon*
- It implies we might need at least  $\log_2(1114112) = 20.08746$  bits for the mapping, ie 3 bytes
  - ie, using a single byte is not enough

# Common Charsets

- **ISO/IEC 8859-1**: using 1 byte, representing up to 256 characters, including Norwegian and Swedish ones, but not full Unicode (eg, no Japanese)
- **UTF-8**: *most used encoding*. Multi-byte representation, up to 4 bytes. Can represent whole Unicode. ASCII (most common) codes need 1 byte, but Norwegian need 2
- **UTF-16**: used internally by Java (eg, “char” variables). Each character takes *at least* 2 bytes. Covers whole Unicode.

# Parsing ISO-8859-1

- As each character is 1 byte, we just read 1 byte (ie 8 bits) at a time
- Direct mapping from 8 bits to a specific character



# Parsing UTF-8

- Read 1 byte at a time, but need to find out if single byte character (eg, “A”), or beginning of multi-byte one (eg, “Ø” or “す”)
- If multi-byte character, need to read all of them before being able to map them to a single char
- Look at first 2 bits in each byte:
  - 0xxxxxxx -> single byte character (using remaining 7 bits)
  - 11xxxxxx -> beginning of a multi-byte character
  - 10xxxxxx -> continuation of a multi-byte character

# Problem

- Reading a bitstring assuming a charset (eg UTF-8) whereas it was encoded in a different one (eg ISO-8859-1)
- You might not see this issue for ASCII codes, as those have same codes in most charsets
- But, as soon as you have non-Latin letters, good luck...
- Example, reading a valid 11xxxxxx 0xxxxxxx from a ISO-8859-1 source would result in a ☐ for the first byte if wrongly displayed as UTF-8 (as 11x... must be followed by 10x...)

HTTP PATCH

# PATCH

- PATCH was introduced in HTTP later on, in its own RFC (5789)
- PUT does a *complete* replacement of a resource
- What if I just want to replace a single field of a resource?
- In PUT, I would have to send the old fields as well, otherwise they would be deleted
  - very inefficient
- PATCH allows to do *partial* changes on a resource

# Wrong PUT on Resource /abc

```
//PUT  
{  
  "foo": 2  
}
```



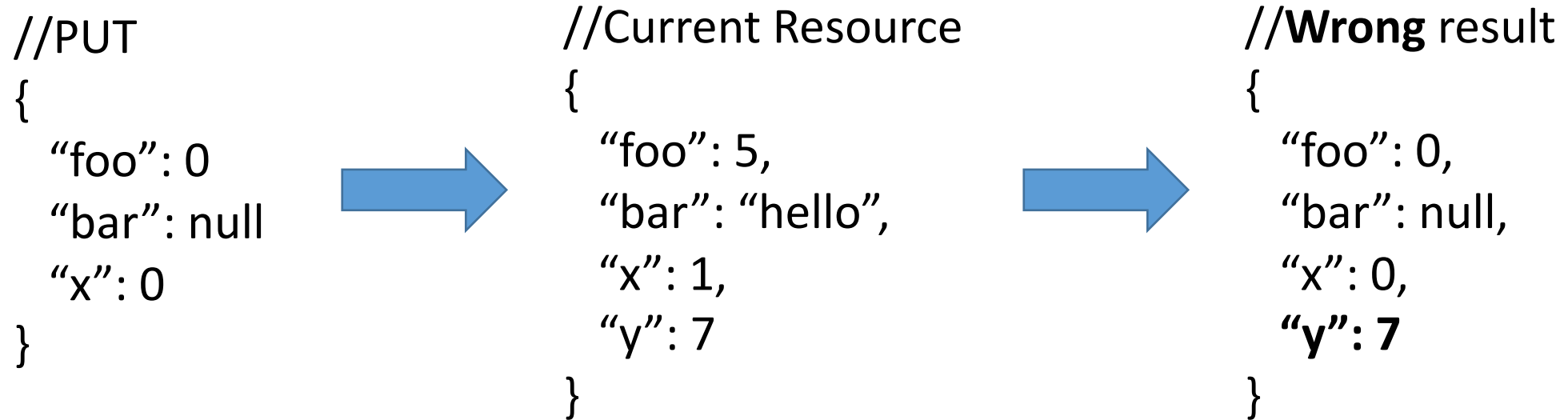
```
//Current Resource  
{  
  "foo": 5,  
  "bar": "hello",  
  "x": 1  
}
```



```
//Wrong result  
{  
  "foo": 2,  
  "bar": "hello",  
  "x": 1  
}
```

- You could implement the PUT this way on the server
- But, besides not following HTTP's semantics, what would be the problem?

# Issue With API Maintenance



- Assume one day you add an *optional* field “y”
- (Old) Clients replacing/creating resource without specifying “y” will wrongly leave current “y” value instead of null

# Partial Updates with PATCH

- PATCH allows you to do *partial* updates
- How to do such updates is NOT specified in HTTP
- A PATCH request will need to specify how to do the update
- For example, could have custom request to increase a numerical field by 1
- A (simple) standard format is *JSON Merge Patch*
  - but there are others, and you can have your custom formats if you need more flexibility

# JSON Merge Patch

- Specified in RFC-7396
- Send a JSON file, and change only fields specified in it

```
//PATCH  
{  
  "foo": 2  
}
```



```
//Current Resource  
{  
  "foo": 5,  
  "bar": "hello",  
  "x": 1  
}
```



```
//Result  
{  
  "foo": 2,  
  "bar": "hello",  
  "x": 1  
}
```



# Tricky: Null vs Undefined

- In JSON, Null and Undefined are two different concepts
- In JSON Merge Patch, setting a variable to null means deleting it, and not changing its value into null

```
//PATCH  
{  
  "foo": null  
}
```



```
//Current Resource  
{  
  "foo": 5,  
  "bar": "hello",  
  "x": 1  
}
```



```
//Result  
{  
  "bar": "hello",  
  "x": 1  
}
```

# Backend Representation

- We send JSON, but the resource might have nothing to do with JSON in the backend
- Example: a row in a SQL database table
- For backend there might be no difference between *null* and *undefined* of a field (eg a column in a SQL table)
- However, tricky when mapping a JSON to a DTO, as a null field has very different semantics from a missing field

# Idempotency

- PUT is idempotent
  - replacing the same resource 1 or n times will result in the same outcome on the server
- PATCH is NOT idempotent
  - if a PATCH adds 1 to a field, repeating it 1 or n times will have very different results
  - note: the effects of JSON Merge Patch are likely idempotent, but the HTTP request itself is not idempotent because using PATCH
  - recall that JSON Merge Patch is just one way to PATCH, and a PATCH could do any kind of modification

# Other Formats

- *JSON Merge Patch* (RFC-7396) is not the only standard format for PATCH, although likely the most common
- Other example: *JSON Patch* (RFC-6902)
  - define list of operations to do on the resource
  - eg: [ { "op": "**test**", "path": "/a/b/c", "value": "foo" }, { "op": "**remove**", "path": "/a/b/c" }, { "op": "**add**", "path": "/a/b/c", "value": [ "foo", "bar" ] }, { "op": "**replace**", "path": "/a/b/c", "value": 42 }, { "op": "**move**", "from": "/a/b/c", "path": "/a/b/d" }, { "op": "**copy**", "from": "/a/b/d", "path": "/a/b/e" } ]
- When talking of HTTP PATCH, always need to define the type format of updates

GUIs

# Web Applications

- In *Web Development and API Design* course we have seen how to build *frontends* with *React*
- *Web Application: frontend + backend*
- *Frontend*: eg, SPA with *React*
- *Backend*: eg, web service like REST

# Static Resources

- HTML files
  - usually just a single *index.html* is SPAs
- CSS
- JavaScript source files
  - eg, *bundle.js*
- Images, documents, or any type of files to download
  - eg, PDFs
- etc.

# Dynamic Resources

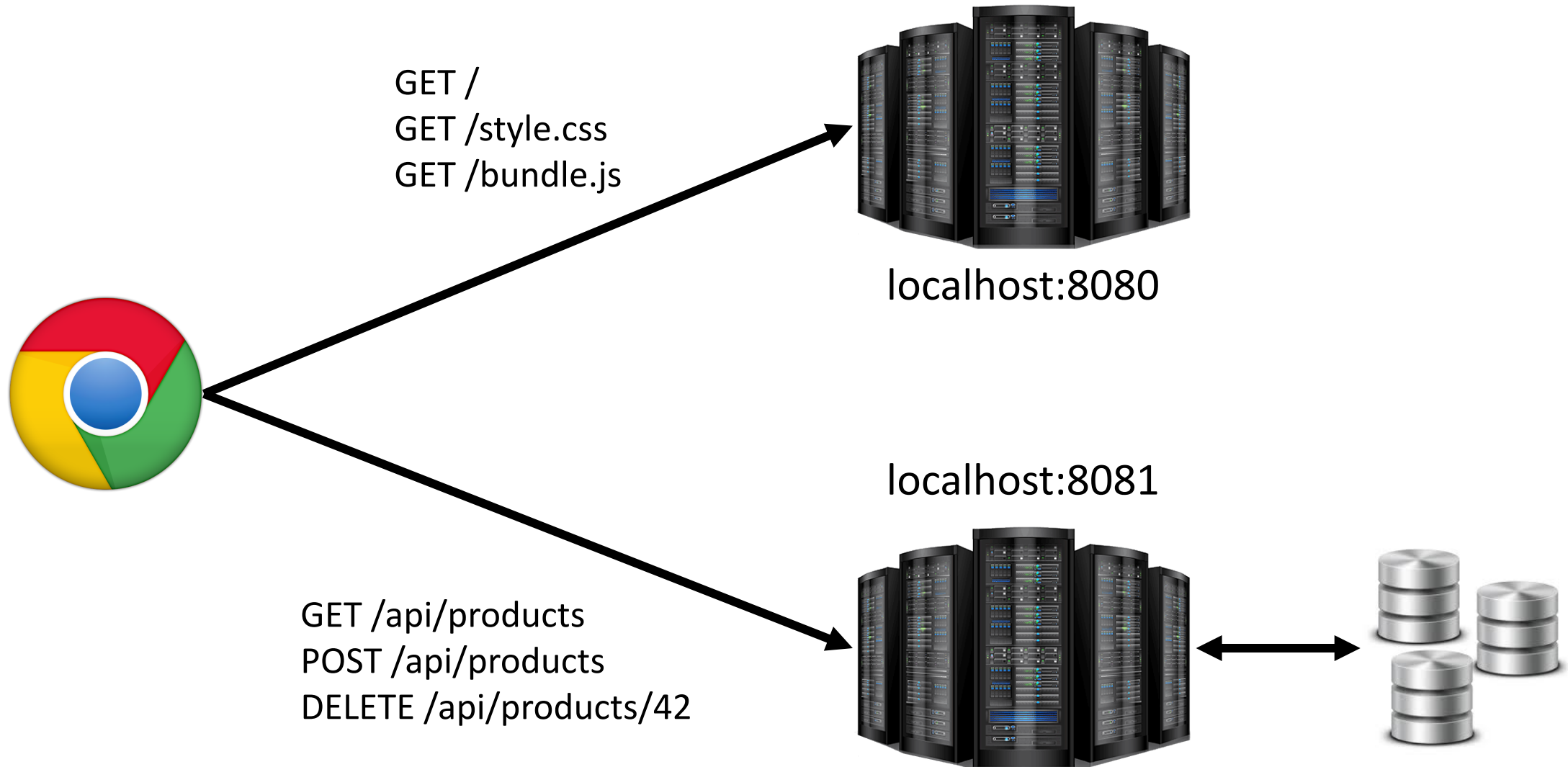
- Data that usually change through time
- Can depend on user interactions
  - create an account, add items to a shopping cart, etc.
- Could have long term storage in SQL and NoSQL databases
- In REST, usually we will get a *representation* of those resource in JSON format
- Handling/generation of dynamic resources usually depend on *business logic* in the so called *backend*



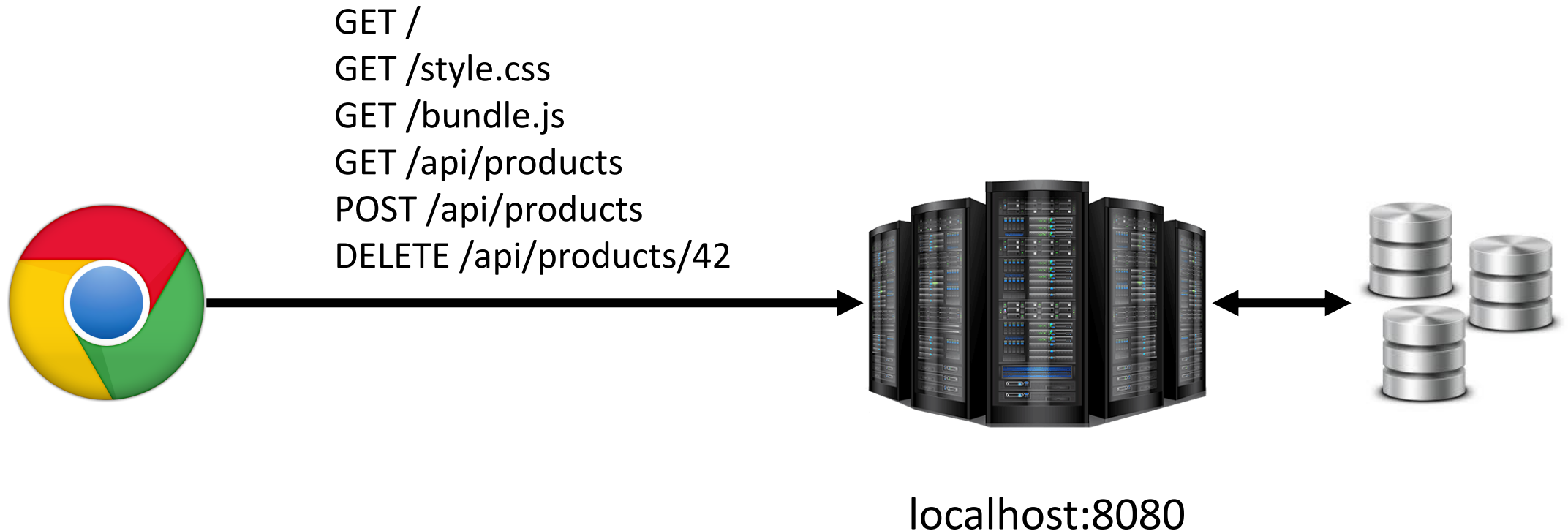
# Static vs Dynamic

- “Usually”, the static resources will define the *frontend* of a web application
  - HTML/CSS/JS/images/etc.
- The *backend* will be a server providing data via JSON, and long term storage with databases
- *Backend* will be a process with business logic written in some programming language (JS, or Java and C#)
- Still, for both static and dynamic resources going to use HTTP

- *Frontend* (e.g., React app) static assets still need to be provided by a HTTP server
- *Backend* (e.g., REST API) with business logic and access to database is still a HTTP server



- From point of view of the browser, no difference between static assets and JSON responses from a REST API
  - still going to use HTTP
- *Frontend* and *backend* can be handled by the same HTTP server



# Web Application in SpringBoot

- By default, *SpringBoot* runs *Tomcat* server
- Files under *resources/static* will be served as static resources over HTTP
- Can call *YARN* from *Maven* to build the *bundle.js*, and copy over directly into *target/classes/static*
  - which will be hence part of the generated *jar* file
- Note: in a microservice, we might want to run NodeJS to handle server-side rendering for SPAs to handle SEO
  - Up to Java 8, that could had been done as well on JVM with *Nashorn*, but it is now deprecated... could be doable though with *GraalVM*

# Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **advanced/rest/charset**
- **advanced/rest/patch**
- **advanced/rest/gui-v1**
- Study RFC-5789 and RFC-7396
- Study relevant sections in RFC-7230 and RFC-7231