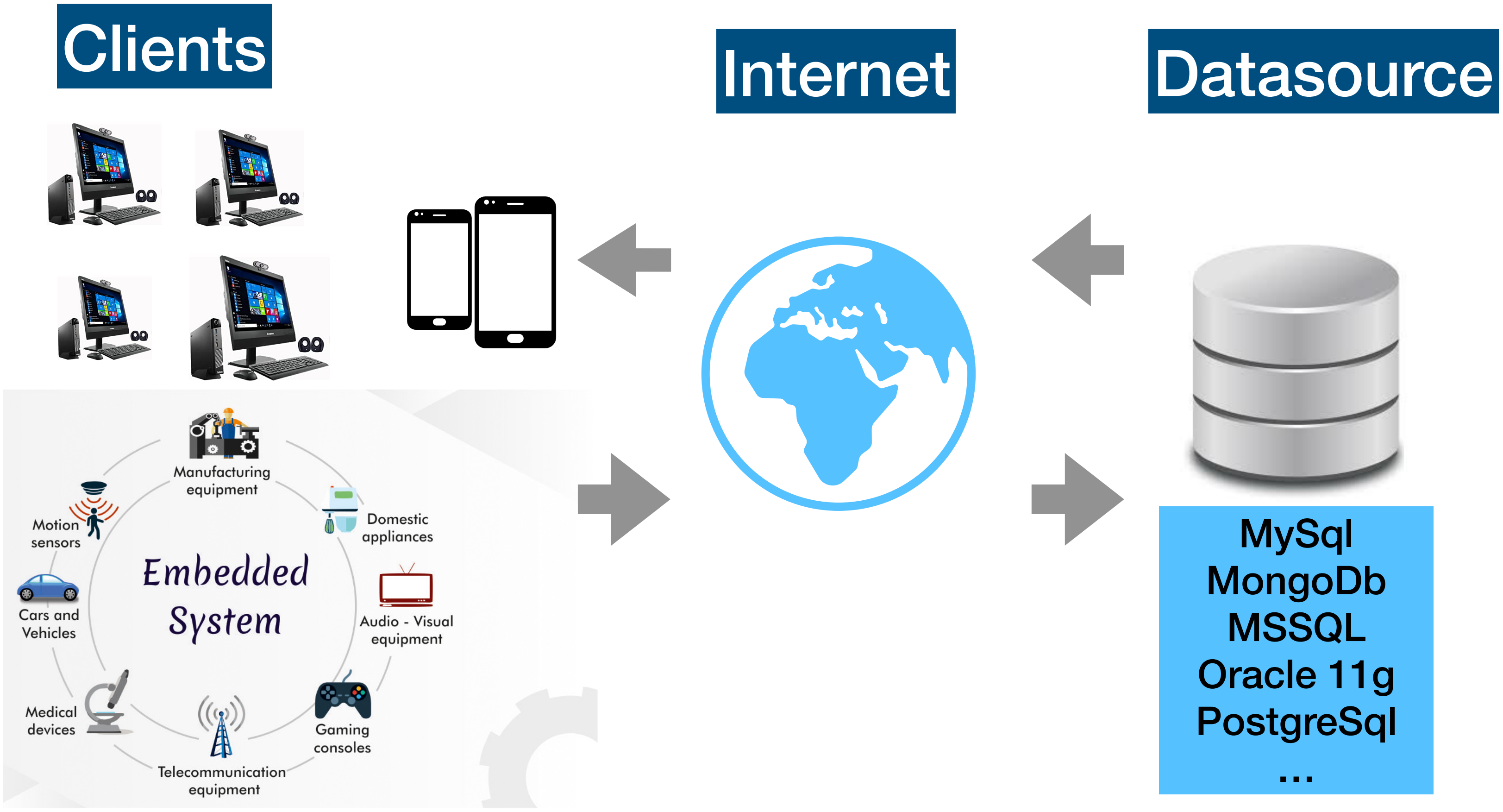
# CS310 Mobile Computing

## Web Programming and Java

PART-1

# The Problem



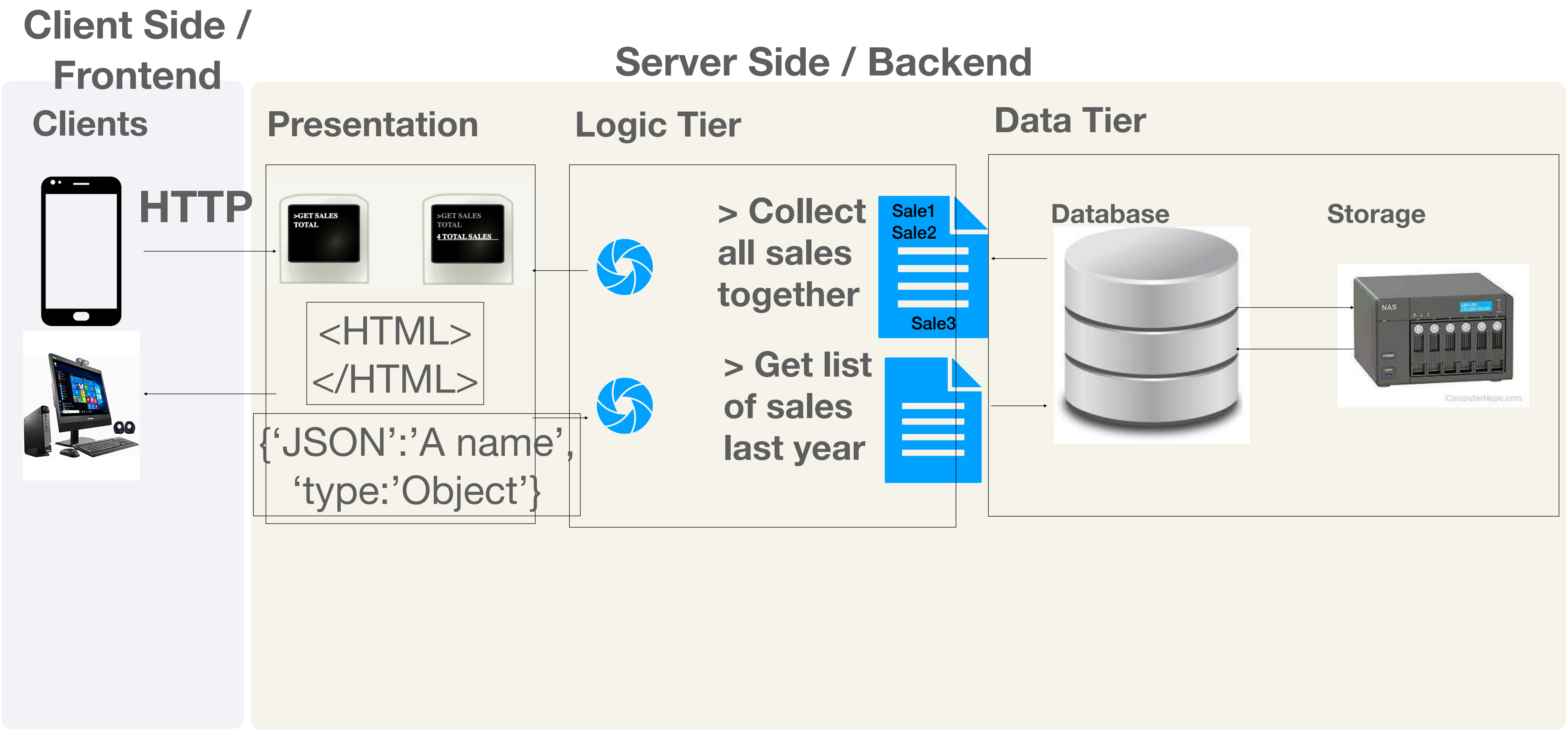
Clients

Internet

Datasource

MySql
MongoDb
MSSQL
Oracle 11g
PostgreSql
...

# Solution: Multi-tier Architecture



Client Side / Frontend

Server Side / Backend

Clients

HTTP

Presentation

<HTML>
</HTML>

{'JSON':'A name', 'type:'Object'}

Logic Tier

> Collect all sales together

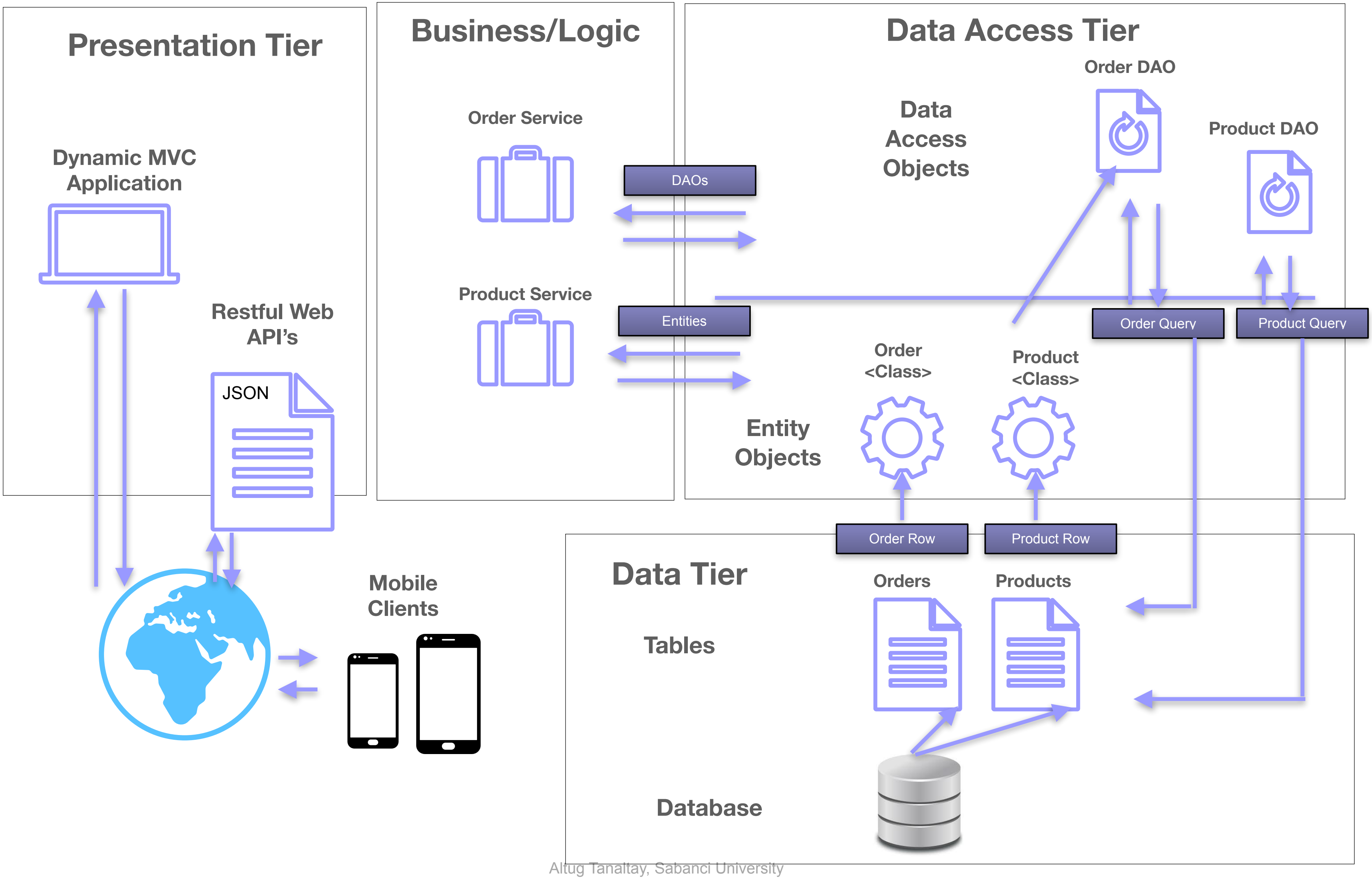
> Get list of sales last year

Data Tier

Database

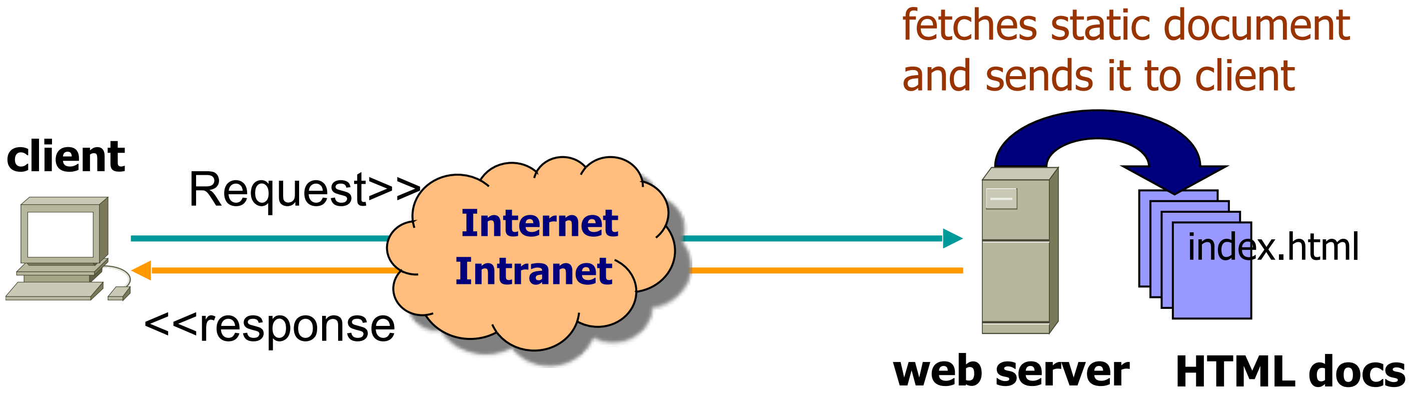
Storage

# Multi-tier Backend

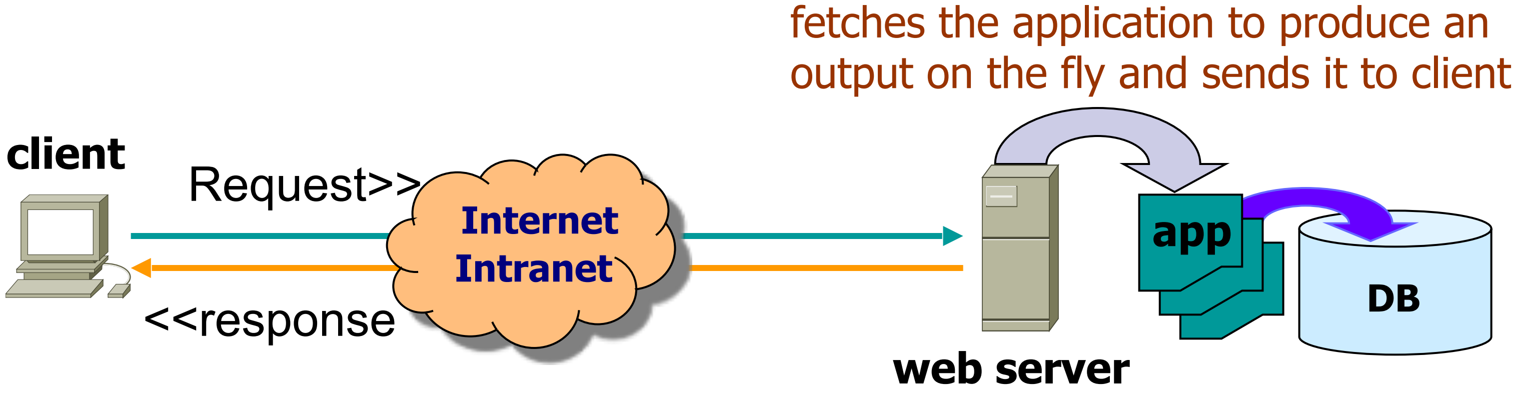


# Static Content vs. Dynamic Content

- Static content

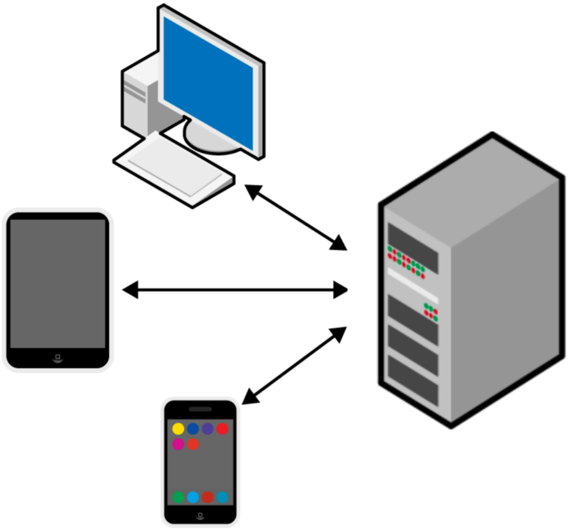


- Dynamic content


fetches the application to produce an
output on the fly and sends it to client
client
Request>>
Internet
Intranet
<<response
app
DB
web server

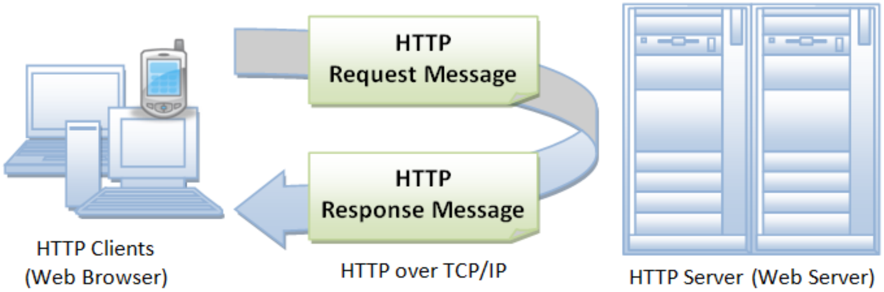
# Clients to Presentation Tier: Client-Server Architecture

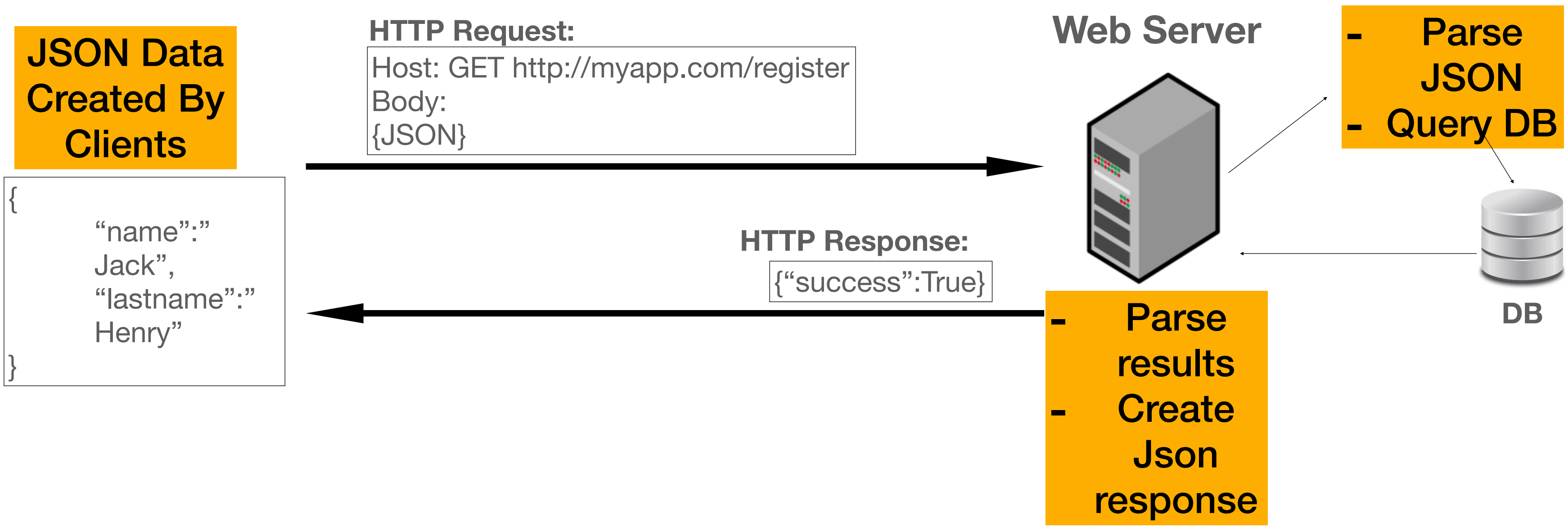
- Architecture of a computer network in which many clients (remote processors) request and receive service from a centralized server (host computer).
- Client computers provide an interface to allow a computer user to request services of the server and to display the results the server returns.
- Servers wait for requests to arrive from clients and then respond to them.
- Ideally, a server provides a standardized transparent interface to clients.



# Client Server Architecture

- The most common protocol for communication is HTTP.
- HTTP Servers (Web Server) are required in order to run  transactions and return responses back.





JSON Data Created By Clients

```
{
        “name”:” Jack”,
        “lastname”:” Henry”
}
```

HTTP Request:

Host: GET http://myapp.com/register
Body:
{JSON}

HTTP Response:

{“success”:True}

Web Server

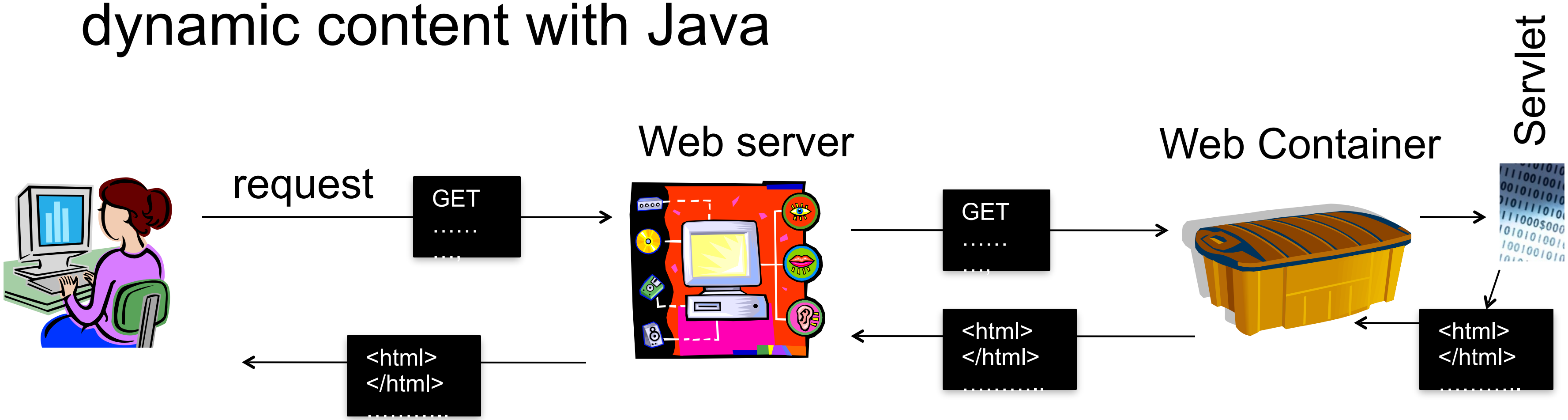
- Parse JSON
- Query DB

DB

- Parse results
- Create Json response

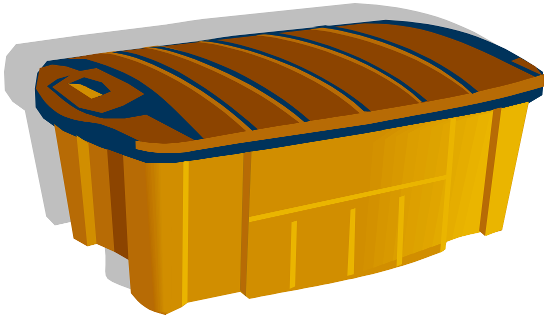
# Are web servers enough for Java?

- To create dynamic content using JavaEE, web servers aren’t enough
- We need Containers in order to host and create dynamic content with Java



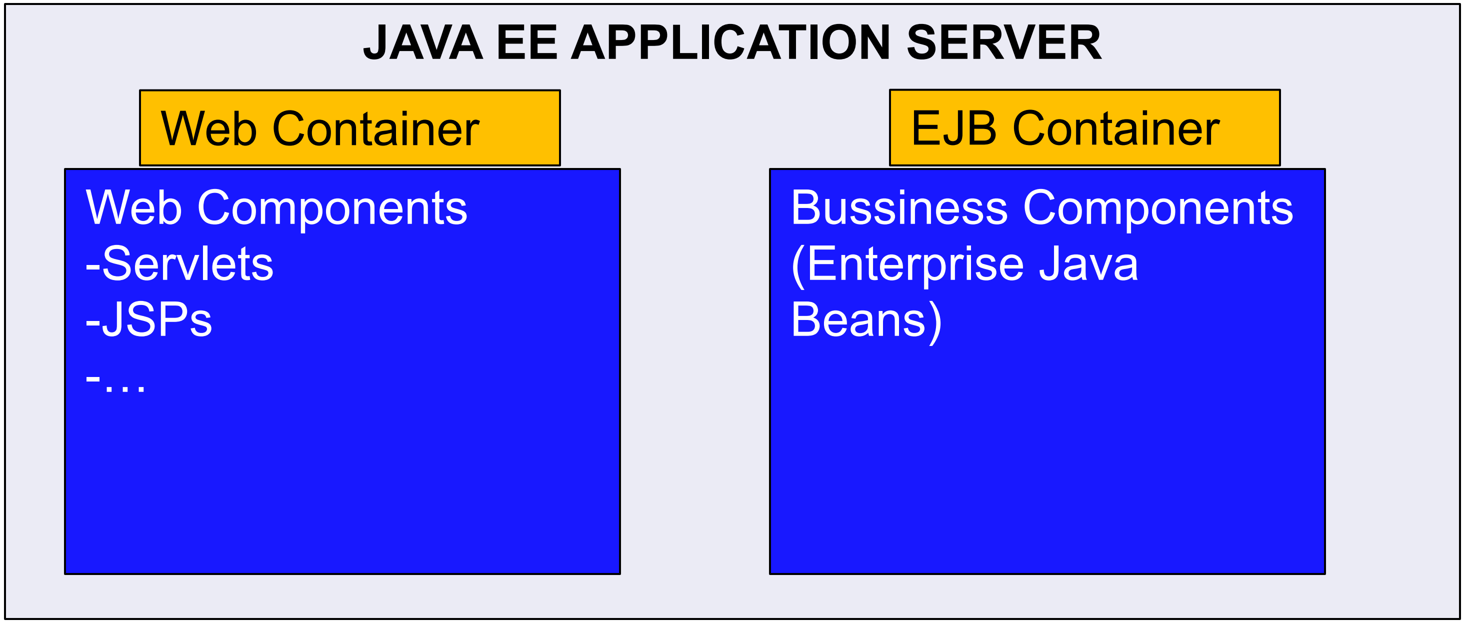
# Web Container

- There is no main method in web applications
- The application is under control of web containers
- Web containers (also called application servers) are like web servers hosting special Java classes
- Web containers provide:
  - Communication support
  - Lifecycle management
  - Multithreading support
  - Declarative security
  - JSP support



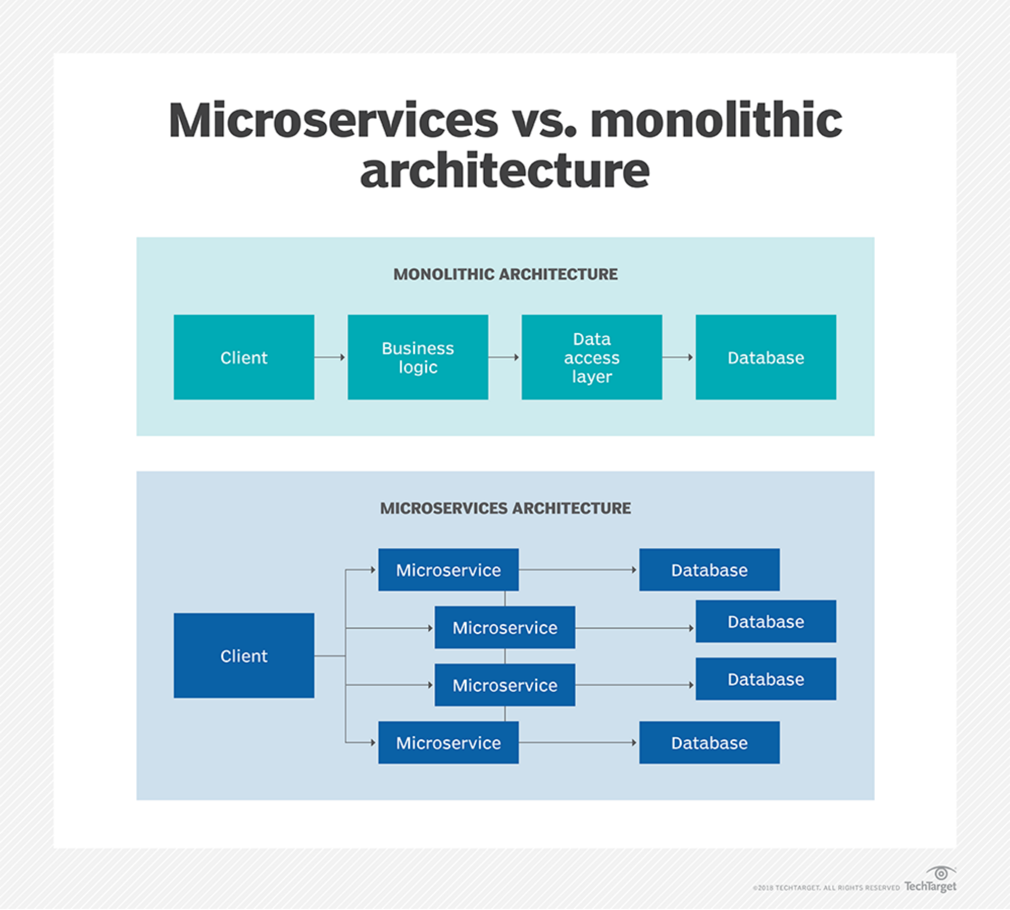
# JavaEE Application Servers

- A full scope JavaEE application server contains both a “web container” and “EJB container”



- Examples. Tomcat (Only web container), Glassfish, Jboss, WebSphere, …

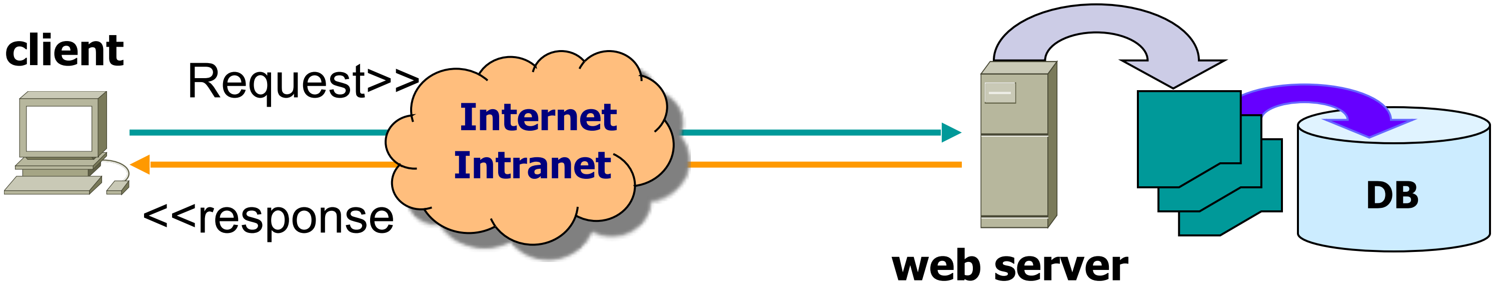
# Design Strategies



- Application Servers force us to create Monolithic design!
- Spring Boot supports micro service architecture: Each subtask can be deployed as a micro service with separate web servers.

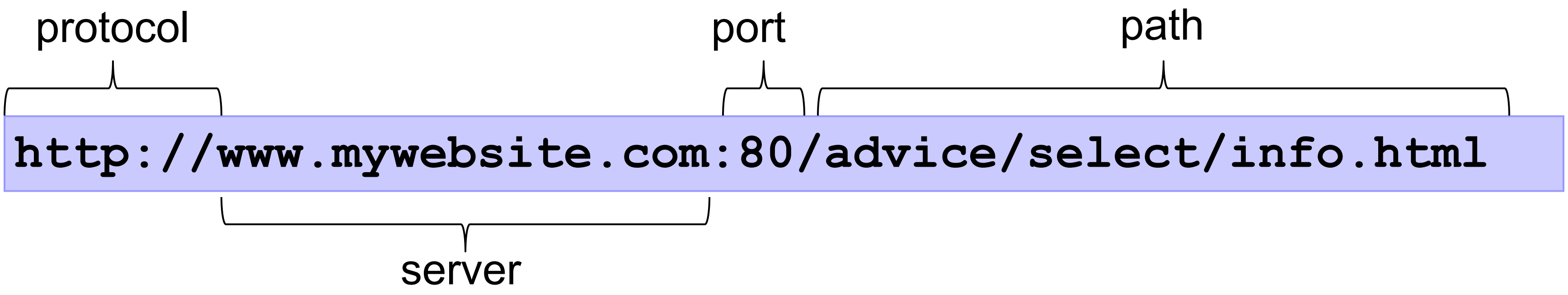
# HTTP Protocol

- Runs over TCP/IP
- TCP is responsible to carry the infomation completely and accurately
- IP is responsible to find the address of host/client
- HTTP carries the information for communication of the host/ client
- HTTP conversation is request and response sentences
- A browser requests and a server responses



# URL – Uniform Resource Locator

- Every resource on the web has its own unique address: URL



protocol

port

path

`http://www.mywebsite.com:80/advice/select/info.html`

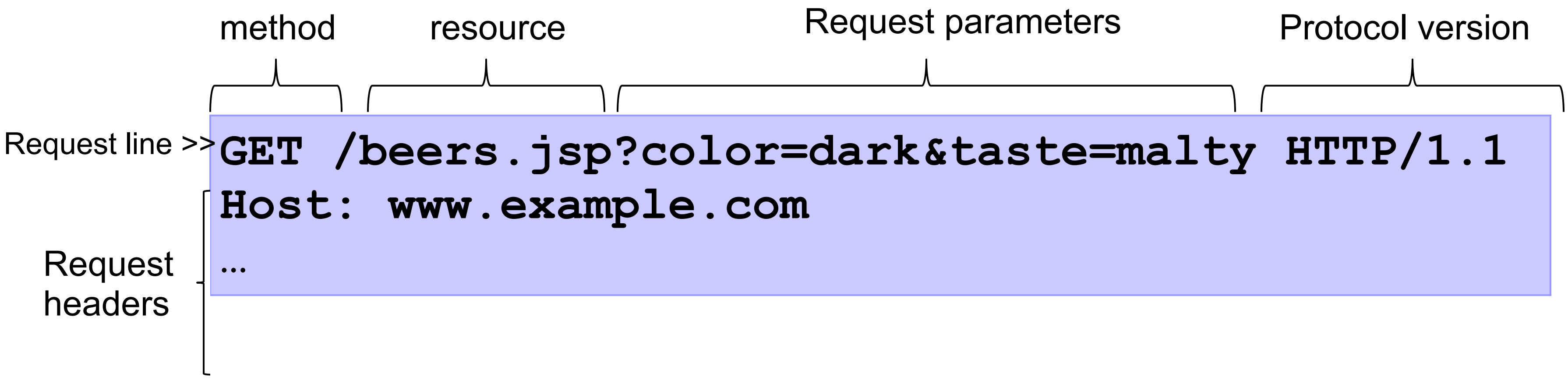
server

# HTTP Request

- The major part is the HTTP method name
  - GET: for simple requests, for getting information
  - POST: for more complex requests, like sending an information to save to the DB
  - Other methods:
    - HEAD, TRACE, PUT, DELETE, OPTIONS, CONNECT

# Anatomy of a GET Request

- Can send information to the server appended to the URL (?name=..&surname=…)
- The total amount of characters in GET are limited, varies according to browser, usually thousands of chars



method

resource

Request parameters

Protocol version

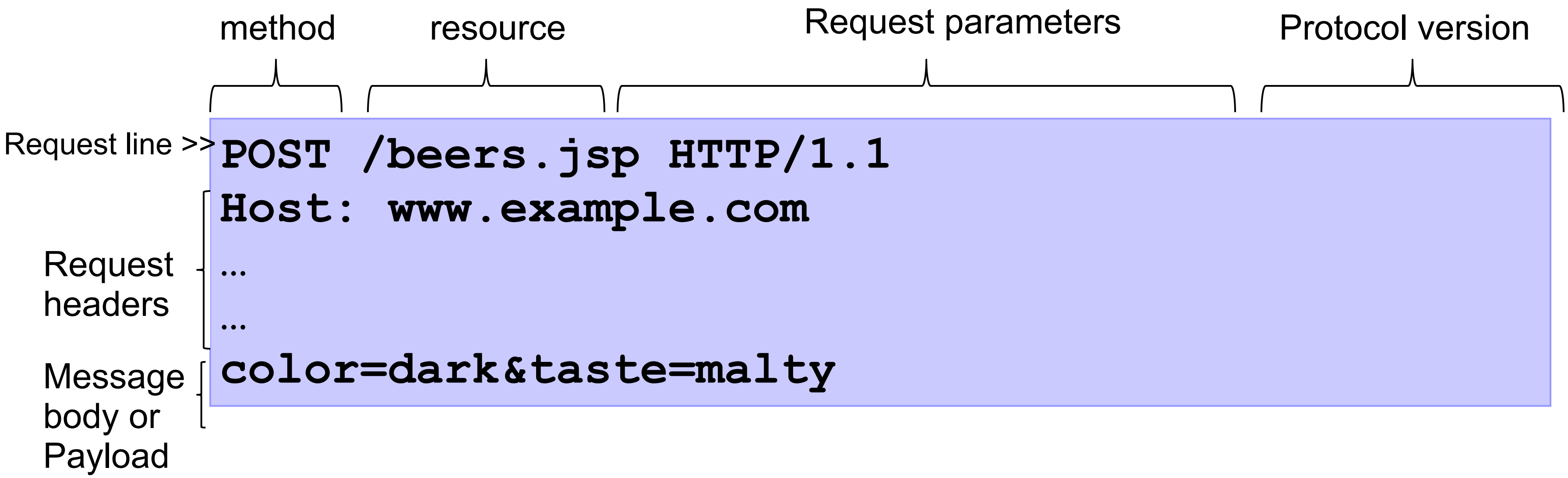
Request line >>

```
GET /beers.jsp?color=dark&taste=malty HTTP/1.1
Host: www.example.com
...
```

Request headers

# Anatomy of a POST Request

- Used for complex requests and data submission
- Request parameters are unlimited and embedded in message body



method

resource

Request parameters

Protocol version

Request line >>

```
POST /beers.jsp HTTP/1.1
Host: www.example.com
…
…
color=dark&taste=malty
```

Request headers

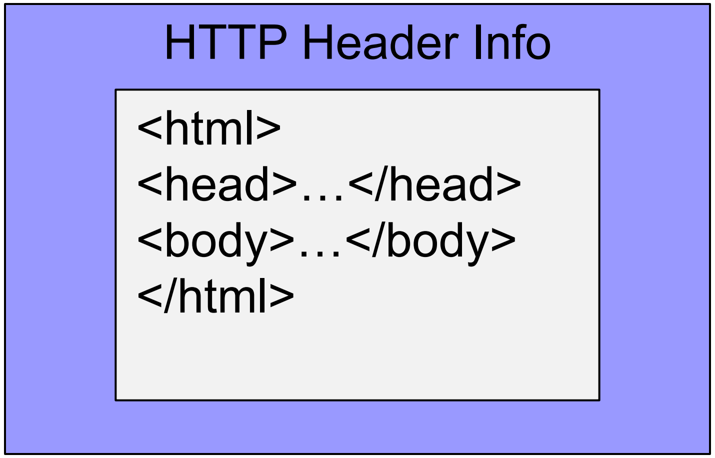
Message body or Payload

# POST vs. GET

| | GET | POST |
|---|---|---|
| **Security** | Less Secure | More Secure |
| **Restrictions on form data length** | Since form data is in the URL and URL length is restricted | No restrictions |
| **Restrictions of form data type** | Only ASCII characters allowed. | No restrictions. Binary data is also allowed. |
| **BACK button / re-submit behaviour** | GET requests are re-executed. | The browser usually alerts the user that data will need to be re-submitted. |
| **Encoding type (enctype attribute)** | application/x-www-form-urlencoded | multipart/form-data or application/x-www-form-urlencoded |

# HTTP Response

- HTML is part of the HTTP response



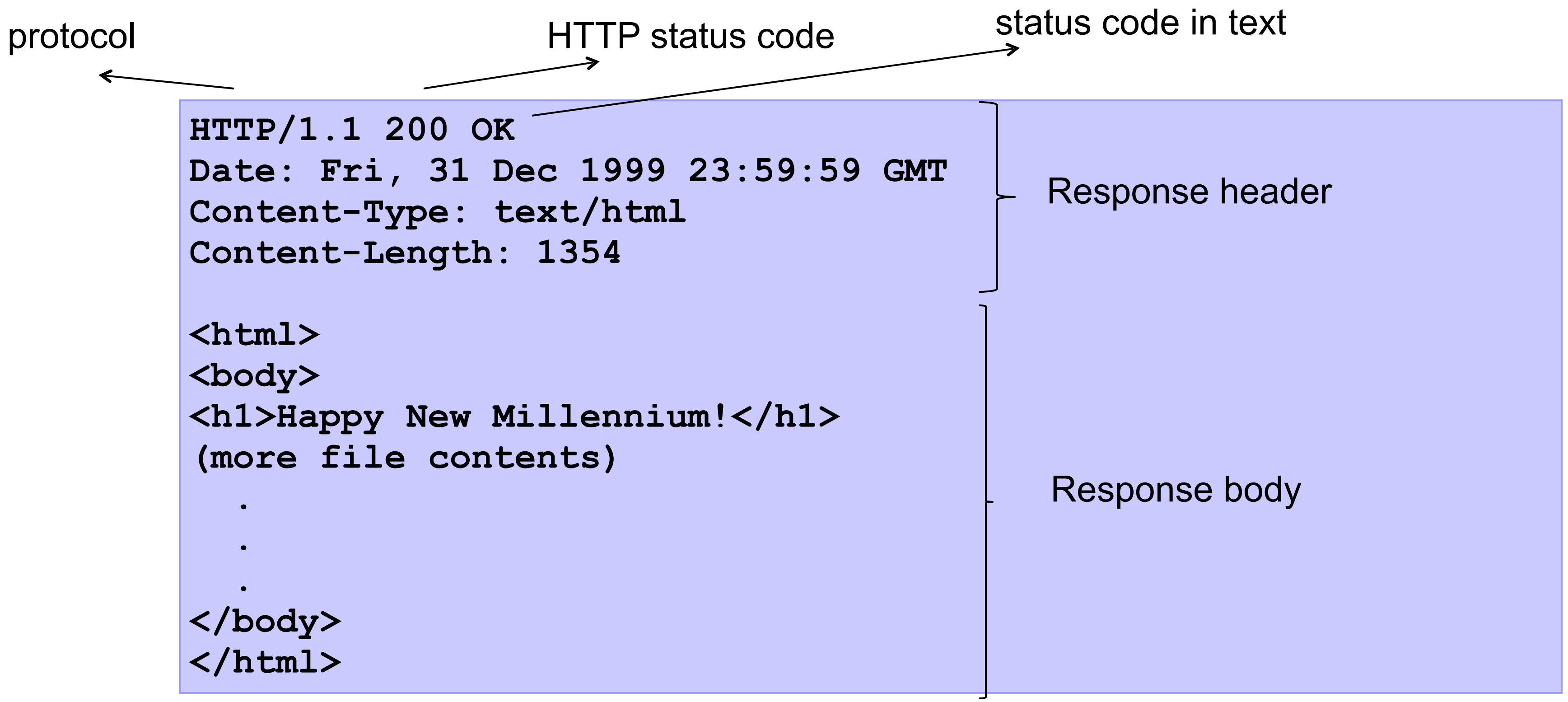
HTTP Header Info

```
<html>
<head>…</head>
<body>…</body>
</html>
```

- Key elements
  - □ A status code (whether the request was successful)
  - □ Content-type (text/html, picture,etc)
  - □ The content (HTML code, image, etc)

# Anatomy of a Response

- An HTTP response has a header and body.
- Header contains information about protocol being used, whether the request was successfull, and the type of content in the body
- Body contains the contents, ex: HTML



protocol

HTTP status code

status code in text

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/html
Content-Length: 1354

<html>
<body>
<h1>Happy New Millennium!</h1>
(more file contents)
  .
  .
  .
</body>
</html>
```

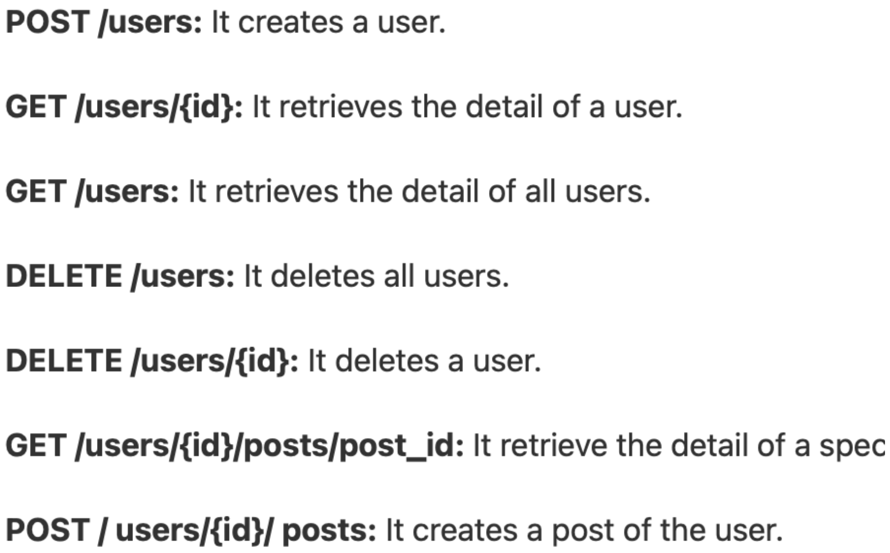
Response header

Response body

# RESTful Web Services

- *Representational State Transfer*, developed by *Roy Thomas Fielding*.
- Uses HTTP with JSON and XML
- The key abstraction is a *resource: Anything can be accessed though URI*

Example URIs



**POST /users:** It creates a user.

**GET /users/{id}:** It retrieves the detail of a user.

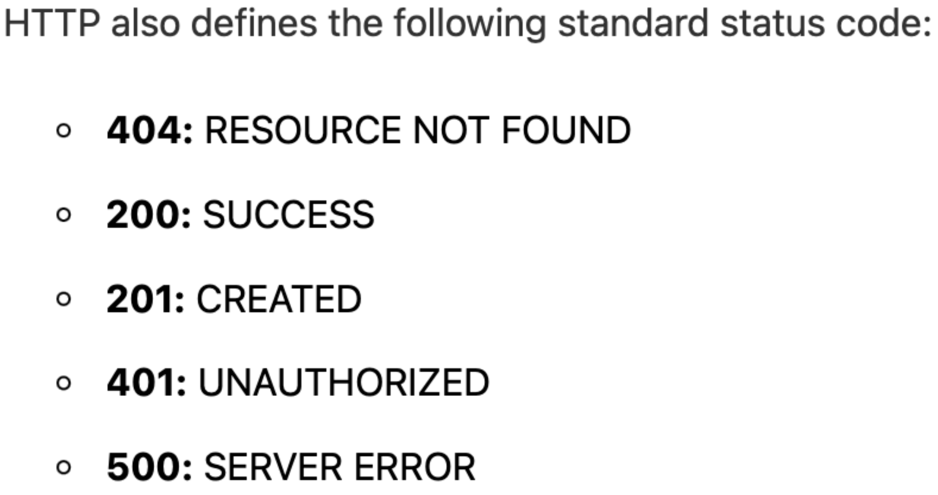
**GET /users:** It retrieves the detail of all users.

**DELETE /users:** It deletes all users.

**DELETE /users/{id}:** It deletes a user.

**GET /users/{id}/posts/post_id:** It retrieve the detail of a spec

**POST / users/{id}/ posts:** It creates a post of the user.



HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

# More about REST

*Constraints*

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.

*Advantages*

- RESTful web services are platform-independent.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like JSON, text, HTML, and XML.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are reusable.
- They are language neutral.

# Configuration - Spring Boot

- To configure a self running Spring Boot Web app create a spring starter project with the following dependencies:

```xml
<dependencies>
       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-web</artifactId>
       </dependency>

       <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-test</artifactId>
          <scope>test</scope>
       </dependency>
   </dependencies>
```

# Configuration - Spring Boot

- In order to make the application deployable as war file, set packaging to “war” in pom.xml and configure a SpringBootServletInitializer:

```
public class ServletInitializer extends SpringBootServletInitializer {

        @Override
        protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
                return application.sources(HelloApiApplication.class);
        }

}
```

class with @SpringBootApplication annotation

# @RestController

- Classes to serve API end points are annotated with @RestController.
- @RequestMapping (class and method level) sets the URI and method for class and method end points

```
package com.server.main;
import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RestControll
er;
//Controller
@RestController
@RequestMapping(“/hello“)
public class HelloWorldController
{
//using get method and hello-world as URI
@GetMapping(path="/hello-world")
public String helloWorld()
{
return "Hello World";
}
}
```

Access with GET request on
http://localhost:8080/hello/hello-world

# HTTP Method Annotations

- @GetMapping
- @PostMapping
- @DeleteMapping
- @PutMapping

# Returning Beans as JSON

- Any Java Bean can be returned from methods.
- Java objects converted to JSON in HTTP message body.
- List of objects are converted to JSON lists.

Controller:

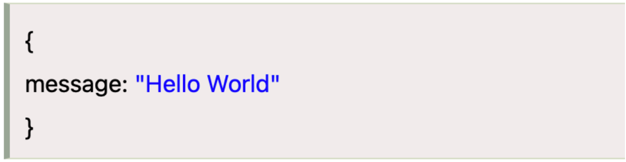
```
//Controller
@RestController
public class HelloWorldController
{
//using get method and hello-world URI
@GetMapping(path="/hello-world")
public String helloWorld()
{
return "Hello World";
}
@GetMapping(path="/hello-world-bean")
public HelloWorldBean helloWorldBean()
{
return new HelloWorldBean("Hello World"); //constructor of HelloWorldBean
}
}
```

Bean:

```
public class HelloWorldBean
{
public String message;
//constructor of HelloWorldBean
public HelloWorldBean(String message)
{
this.message=message;
}
//generating getters and setters
public String getMessage()
{
return message;
}
public void setMessage(String message)
{
this.message = message;
}
@Override
//generate toString
public String toString()
{
return String.format ("HelloWorldBean [message=%s]", message);
}
}
```

It returns the message "**Hello World**" in JSON format.

Launch the **HelloWorldController**. The URL of the browser changes to **localhost:8080/hello-world-bean**

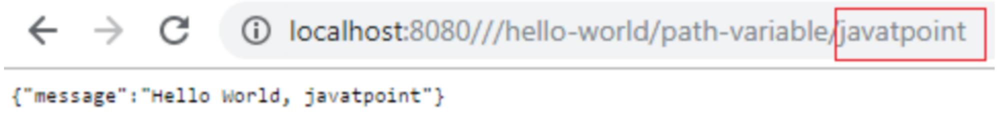


```
{
message: "Hello World"
}
```

# Annotation: @PathVariable

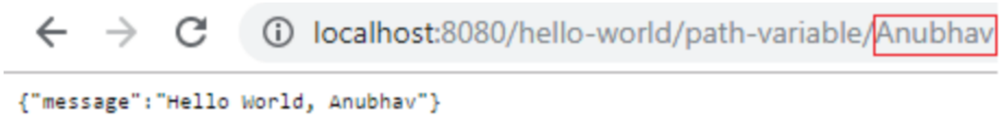
- The @PathVariable annotation is used to extract the value from the URI.

```
//passing a path variable
@GetMapping(path="/hello-world/path-variable/{name}")
public HelloWorldBean helloWorldPathVariable(@PathVariable String name)
{
return new HelloWorldBean(String.format("Hello World, %s", name)); //
%s replace the name
}
```



localhost:8080///hello-world/path-variable/javatpoint

```
{"message":"Hello World, javatpoint"}
```



localhost:8080/hello-world/path-variable/Anubhav

```
{"message":"Hello World, Anubhav"}
```

# Consuming JSON Data: @RequestBody

- The *@RequestBody* annotation maps body of the web request to the method parameter.
- The body of the request is passed through an HttpMessageConverter. It resolves the method argument depending on the content type of the request.

```
//method that posts a new user detail
@PostMapping("/users")
public void createUser(@RequestBody User user)
{
User sevedUser=service.save(user);
}
```

```
class User{
	String username;
	String password;

	//getters & setters
}
```

In order to access the end point, create a POST request with JSON in request body:

```
{“username”:”usr”,
“password”:”pass”}
```

# Throwing Exceptions to HTTP Response

- Extend from an exception type, annotate the class with @ReponseStatus, threw the exception in controller methods:

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;
@ResponseStatus(HttpStatus.NOT_FOUND)
public class UserNotFoundException extends RuntimeException
{
public UserNotFoundException(String message)
{
super(message);
}
}
```

```
@GetMapping("/users/{id}")
public User retriveUser(@PathVariable int id)
{
User user= service.findOne(id);
if(user==null)
//runtime exception
throw new UserNotFoundException("id: "+ id);
return user;
}
```

# For More Info …

- https://www.javatpoint.com/restful-web-services-tutorial
- https://www.baeldung.com/rest-with-spring-series
- Spring Official: https://spring.io/guides/tutorials/rest/