Veebirakenduse projekt (ITI0302)

Loeng 1

Siim Rebane

The internet protocol stack

- Internet layer (IPv4, IPv6)
  - Handles that message gets from point A to point B
  - IP header includes source address and destination address
- Transport layer (TCP, UDP)
  - TCP/UDP are within IPv4 or IPv6 packet
  - TCP and UDP add ports (port numbers: 0–65535)
    - 0–1023 — well known / privileged ports (need admin rights in UNIX systems)
    - 1024–49151 — regular ports, can use any of them
    - 49152–65535 — dynamic/ephemeral ports (OS assigns these for outgoing connections)
  - For example source port is a random port on your machine
  - Destination port: 80 (HTTP), 443 (HTTPS)
  - This lets multiple applications share one IP address

HTTP(S)

- Application layer (HTTP, HTTPS)
  - HTTP is wrapped in TCP message (TCP payload is HTTP message)
  - HTTP is the actual protocol used in web
  - HTTPS is encrypted (TLS/SSL) HTTP
  - HTTP communicates via request-response pattern
  - Client sends request → Server sends response
  - Stateless protocol (each request is independent)
  - Default ports: HTTP (80), HTTPS (443)

HTTP protocol structure

Sample request:

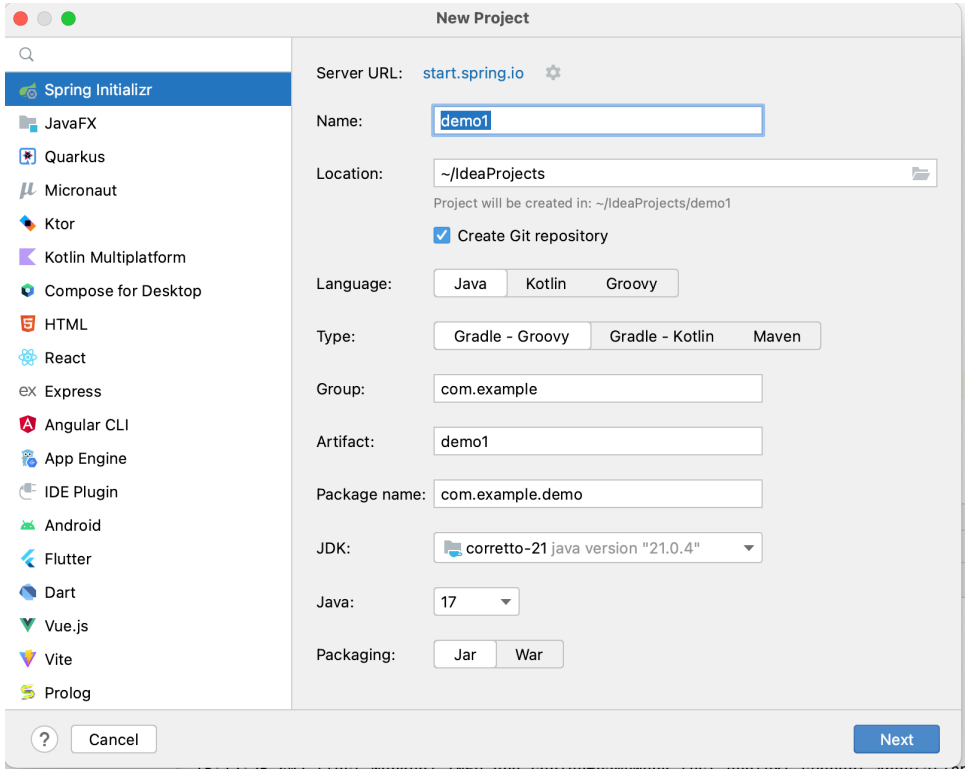
```
GET /api/books/123 HTTP/1.1          ← method + path + HTTP version
Host: localhost:8080                 ← header (automatically added)
Accept: application/json             ← header (expected content type)
Authorization: Bearer token123       ← header (auth information)
                                     ← empty line (required)
                                     ← request body (currently missing)
```

Sample response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 54
                                     ← empty line (required)
{"id": 123, "title": "Clean Code", "author": "Martin"}   ← response body
```

Initialization of backend project

- https://start.spring.io/
- IntelliJ: File → New → Project
- Choose Spring Initializr



Maven / Gradle

- Build automation tool
- Dependency management
- Library repository (https://mvnrepository.com/)
- Maven: `pom.xml`
- Gradle: `build.gradle`

Other options

- Language (we use Java)
- Spring Boot (use default = latest stable version)
- Java versions
  - LTS (long term support) versions are: 8, 11, 17, 21, 25
  - Usually, latest LTS is a good choice (currently 25)
  - You may use the latest version

Project metadata

- Name — project name
- Description — description
- Package name — base package of your application
- From https://maven.apache.org/guides/mini/guide-naming-conventions.html:
  - *groupId* uniquely identifies your project across all projects. A group ID should follow Java's package name rules. This means it starts with a reversed domain name you control.
  - *artifactId* is the name of the jar without version.

JAR / WAR

- JAR — Java Archive
- WAR — Web Application Resource
- Both are actually zip files
- WAR file contains a web application that can be deployed on an installed server
- JAR file can be run without a server (`java -jar test.jar`)

Dependencies

- In the dependencies tab we can add other libraries that are automatically downloaded
- For starters we only need Spring Web (will add more later)
- Maven caches downloaded dependencies in `.m2` folder
- Gradle caches downloaded dependencies in `.gradle` folder

Run project

- Run generated project main method
- You should see something like that:

```
INFO --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (http)
INFO --- [main] com.example.demo.DemoApplication        : Started DemoApplication in 1.2 seconds
```

Add Hello World controller

```
@RestController
public class HelloController {

    @GetMapping("/")
    public String helloWorld() {
        return "Hello World";
    }
}
```

- Rerun application (you need to rerun after each change)
- Go to localhost:8080

REST — Representational State Transfer

- Architectural style with formal constraints (not protocol, not standard)
- Academic definition says:
  - Client/Server
  - Stateless
  - Cache
  - Uniform Interface
  - Layered System
  - Code on Demand
- In practice for web APIs, we focus on:
  - Stateless (each request is independent)
  - Uniform Interface (consistent URL patterns)
  - Client/Server (obvious — that's what we're building)

REST — resource

- A **resource** is any piece of data you can name:
  - User, Book, Order, Comment, Product
  - Resources are NOUNS, not actions
  - Each resource has a unique identifier (URL)
- Examples:
  - User with ID 123 → `/users/123`
  - All users → `/users`
  - Order 456 → `/orders/456`

REST URL rules

- ALWAYS use plural nouns (even for single items)
  - ✅ `GET /users/123` (one user)
  - ✅ `GET /users` (all users)
  - ❌ `GET /user/123` (wrong!)
- Collections vs single resource:
  - `/users` → collection (array)
  - `/users/123` → single resource (object)
- No verbs in URLs:
  - ❌ `/getUsers`, `/createUser`, `/deleteUser`
  - ✅ `GET /users`, `POST /users`, `DELETE /users/123`

REST hierarchies

- Nested resources (parent-child relationships)
  - `/users/123/posts` → posts belonging to user 123
  - `/posts/456/comments` → comments on post 456
  - Keep it shallow (2–3 levels max):
    - ✅ `/users/123/posts`
    - ❌ `/users/123/posts/456/comments/789/likes`

HTTP methods = actions on resource

HTTP Method	Action	Body
GET	Read	No
POST	Create	Yes
PUT	Update/Replace	Yes
PATCH	Update/Modify	Yes
DELETE	Delete	No

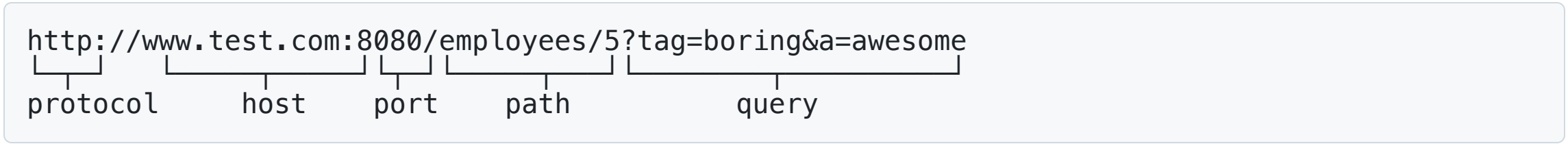
HTTP status codes

- Response code is returned to the client by the server to signify the final status of the request
- The first digit signifies the main category of response code
  - For example when POST request creates a resource then you should return 201
  - If you return 200, it's not wrong but you are giving less information about the state
  - On the following slide there is a list of codes we should be aware of and use in this course

HTTP status codes

Status Code	Meaning	
100	Informational	Rarely used, ignore
200	OK	Success
201	Created	New resource created (POST)
204	No content	Often used with DELETE
3xx	Redirect	Resource is elsewhere — e.g. 302 sends the browser to another URL
400	Bad Request	Client sent information that server can't handle
401	Unauthorized	You are not logged in
403	Forbidden	You have no rights to access
404	Not Found	No page / resource on that URL
500	Internal Server Error	Bug in backend code

URL



- Protocol — http (plain text) / https (encrypted)
- Host — comes from DNS and server configuration
- Port — if no port specified then uses protocol's default (http = 80, https = 443)
- Path — path is divided by slashes and controlled in Java
- Query — starts from `?`, query parameters are separated by `&`

Defining controllers

- All classes must be inside your project base package
- All classes that contain code that can be called from web have to have `@RestController` annotation
- All methods have to have a mapping annotation (e.g. `@GetMapping`, `@PostMapping`, ...)
- Within the mapping annotation you must define the request path

Path variable

- You can read path variables by encapsulating them with braces in the mapping annotation

```
@GetMapping("/employees/{id}")
public Long test(@PathVariable("id") Long id) {
```

REST query parameter best practices

- Query parameters are for filtering/searching collections
- ✅ Good uses of query params:
  - `/posts?status=draft&author=john`
  - `/users?role=admin&active=true`
  - `/products?minPrice=10&maxPrice=100`
  - `/posts?page=2&limit=20`
- ❌ Don't use query params for resource identification:
  - `/users?id=5` (wrong — use `/users/5`)
- Path vs query:
  - Path identifies WHAT: `/users/123/posts`
  - Query filters HOW: `/posts?status=published&sort=date`

Request param

- You can use `@RequestParam` to extract query parameters
- `/employees?employeeId=5&testId=4`

```
@GetMapping("/employees")
public Result test(@RequestParam("testId") Long id) { …. }
```

- NB: request params are not part of the path and not defined within the mapping annotation

JSON sample

```
{
    "firstName": "Siim",
    "lastName": "Rebane",
    "position": "Software Developer",
    "vacationDays": [
        { "from": "2026-07-01", "to": "2026-07-14" },
        { "from": "2026-12-23", "to": "2026-12-31" }
    ]
}
```

JSON

- Objects are enclosed in braces `{}`
- Use colon `:` to separate key-value pairs
- Add comma between each key-value pair
- Arrays are enclosed in brackets `[]`
- Add comma between array elements
- 7 value types: string, number, object, array, true, false, null
- Validate JSON: https://jsonlint.com/
- If you return an object from a controller it will be converted to JSON

DTOs

- DTO — data transfer object: carries data between client and server, no logic
- Recommended: use a **record** — one line, and you get constructor, `equals`, `hashCode`, `toString` for free:

```
public record BookDto(Long id, String name, Integer pageCount) {}
```

- Works automatically with `@RequestBody` and as a return value
- NB: accessors are `book.name()`, not `book.getName()`

DTOs — mutability

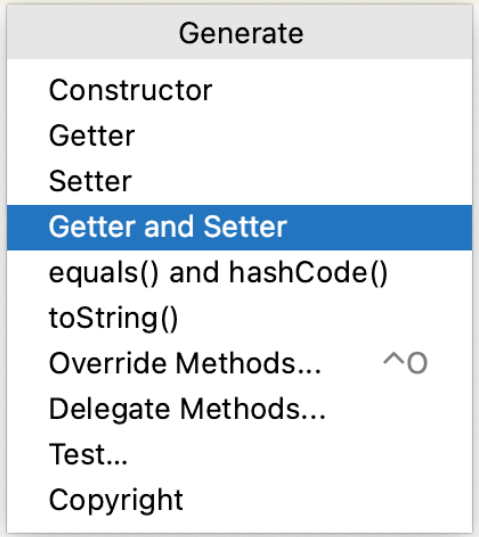
- **Mutable** = can be changed after creation; **immutable** = values are fixed once created
- A record is immutable — perfect for DTOs: a request that arrived should not be modified afterwards
- A classic class with getters and setters is the **mutable** alternative — also fine for DTOs, and you will see it in a lot of existing code
- When you NEED mutability, you need a class — e.g. JPA entities (week 4) cannot be records

DTO — the classic class

- Getter and setter methods can be auto-generated by IntelliJ:

```
no usages   new *
public class BookDto {
    no usages
    private String name;
    no usages
    private String author;


}
```



@RequestBody

- The same way we can output data, the server can read data from the request body
- Annotate the variable where you want to receive the sent body
- For input use DTOs

```
@PostMapping("test")
public Result test(@RequestBody Request request) {
}
```

Status codes

- In case of success, if we want to respond with some other status code we can use `@ResponseStatus`
- In case of failure you can throw `ResponseStatusException`

```
@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteBook(@PathVariable long id) { … }
```

```
throw new ResponseStatusException(HttpStatus.NOT_FOUND, "Book not found");
```

Status codes (more complex)

- You can also take full control using `ResponseEntity<T>`
- Define the method to return `ResponseEntity<T>` — then you choose the status per response:

```
@PutMapping("/{id}")
public ResponseEntity<Book> updateBook(@PathVariable long id, @RequestBody Book book) {
    if (bookExists(id)) {
        return ResponseEntity.ok(book);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Testing your API

- Typing a URL in the browser only sends GET — for POST/PUT/DELETE you need a tool
- IntelliJ HTTP client — create a file ending in `.http`, press the green arrow:

```
POST http://localhost:8080/employees
Content-Type: application/json

{ "firstName": "Siim", "lastName": "Rebane" }
```

- `curl` works everywhere:

```
curl -X POST localhost:8080/employees \
  -H "Content-Type: application/json" -d '{ "firstName": "Siim" }'
```

- Postman is a popular GUI alternative
- Later we add OpenAPI/Swagger, which generates a test UI from your code