Toegepaste Informatica

1TX

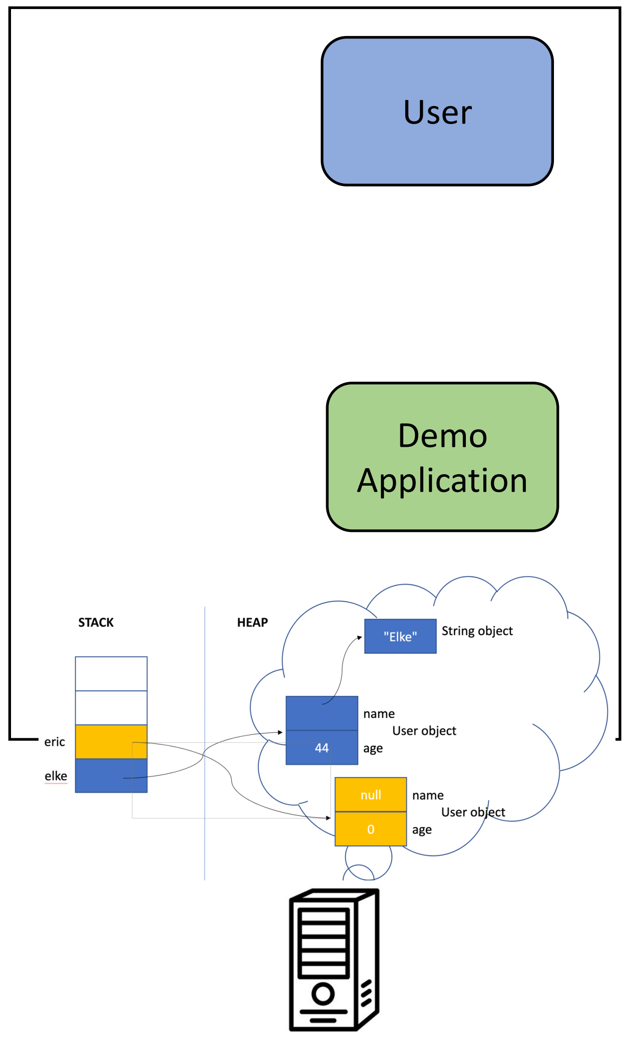
2022-2023

Back-End Development

REST API – GET REQUEST

G. Jongen, J. Pieck, E. Steegmans, B. Van Impe

RECAP



Spring Boot

Tomcat

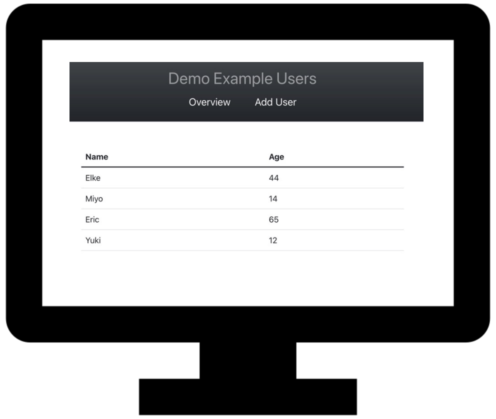
Application Server
Tomcat

- is a server that hosts applications or software that delivers a business application through a communication protocol (e.g. HTTP)
- exposes business logic to the clients, which generates dynamic content
- an application server framework includes software components available to a software developer through an application programming interface (API)

Framework
Spring Boot

- is an application server framework
  - includes software components available to a software developer through an application programming interface (API)
- is an easy to get-started addition to the Spring framework
  - Spring is a
    Java framework that makes programming Java quicker, easier and safer
- makes it easy to create stand-alone, production-
  grade Spring based applications that you can "just run" with minimal or zero configurations
  - avoids a lot of boilerplate code
  - hides a lot of complexity behind the scene

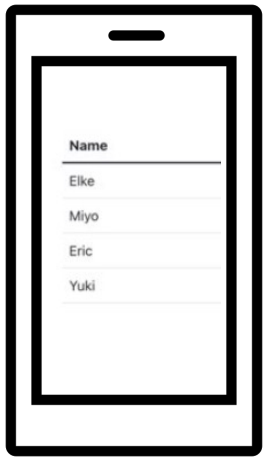
END GOAL



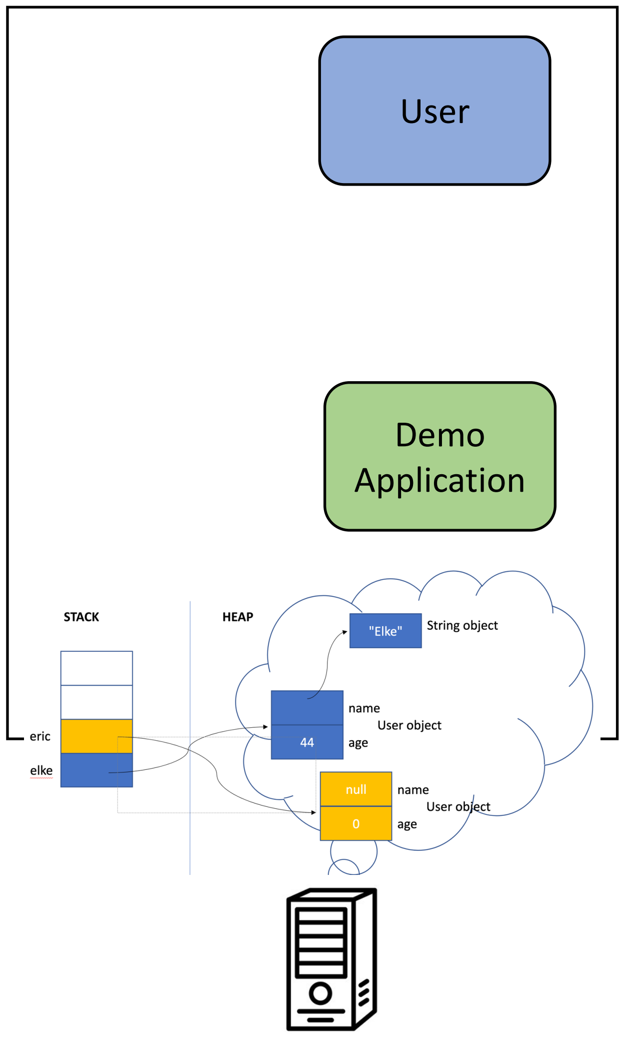
Demo Example Users

Overview Add User

Name	Age
Elke	44
Miyo	14
Eric	65
Yuki	12

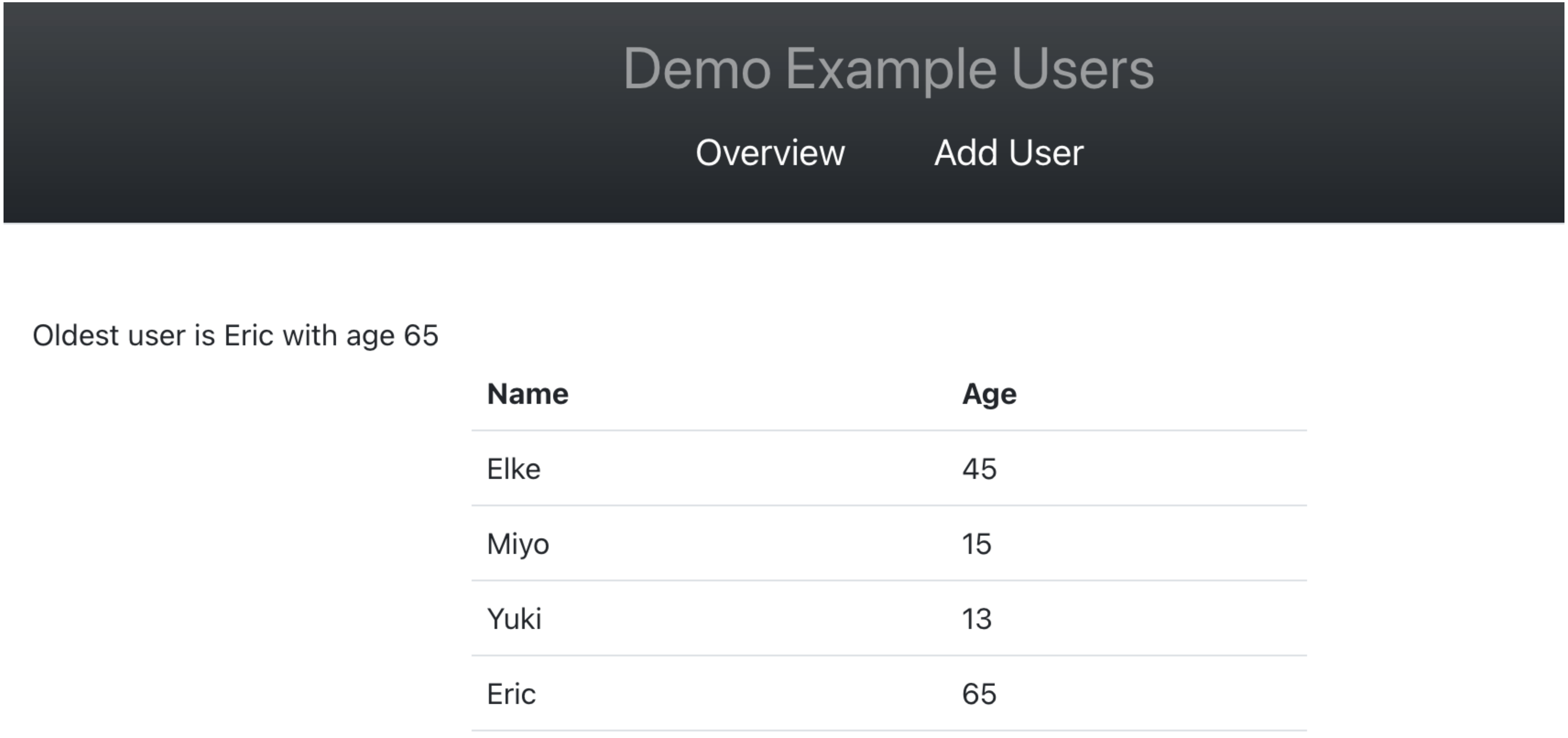


Name
Elke
Miyo
Eric
Yuki



Spring Boot

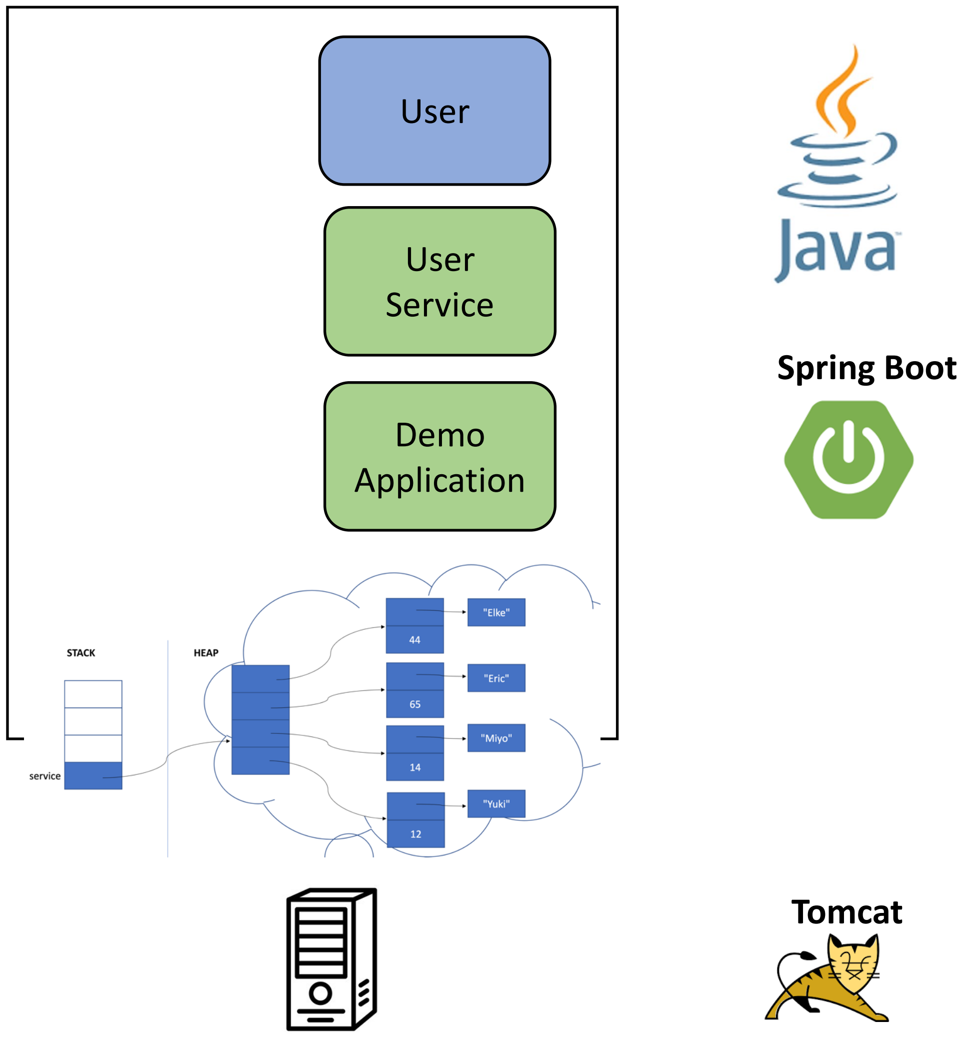
Tomcat



Oldest user is Eric with age 65

Name	Age
Elke	45
Miyo	15
Yuki	13
Eric	65

END GOAL - STEP 1

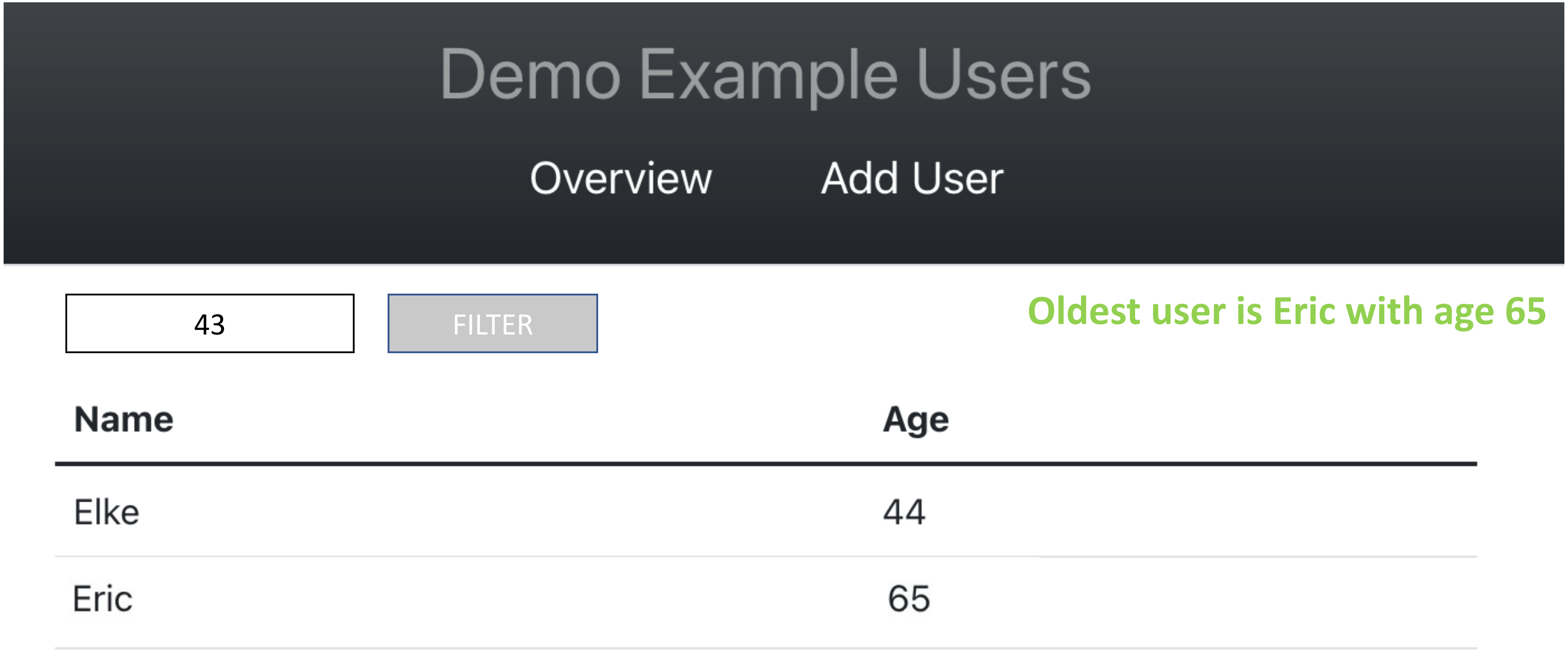


@Service

- Service Components are
  - The glue that you need between your objects and the functionality that is wanted by the rest controller
  - the class file which contains @Service annotation

```
@Service
public class UserService {

  private List<User> userRepository
    = new ArrayList<>();

  public UserService() {}

  public List<User> getAllUsers() {
    return userRepository;
  }

  public boolean add(User user) {
    return userRepository.add(user);
  }
}
```

```java
@Service
public class UserService {

  private List<User> userRepository = new ArrayList<>();

  public List<User> getAllUsers() {
    return userRepository;
  }

  public User getOldestUser() {
    User oldest = null;
    if (userRepository.size()>0) {
      oldest = userRepository.get(0);
      for (User user : userRepository) {
        if (user.getAge() > oldest.getAge())
          oldest = user;
      }
    }
    return oldest;
  }
}
```



Oldest user is Eric with age 65

Name	Age
Elke	44
Eric	65

```java
public List<User> getUsersWithAgeOlderThan(int age) {
  List<User> users = new ArrayList<User>();
  for(User user: userRepository) {
    if (user.getAge() > age)
      users.add(user);
  }
  return users;
}
```

YOU CAN ALSO USE JAVA STREAMS

```java
public List<User> getUsersWithAgeOlderThan(int age) {
  return userRepository.stream().filter(user -> user.getAge()>age).toList();
}
```

Java Streams

- Java streams enable functional-style operations on streams of elements.
- A stream is an abstraction of a non-mutable collection of functions applied in some order to the data.
- A stream is not a collection where you can store elements.

```java
public List<User> getUsersWithAgeOlderThan(int age) {
  List<User> users = new ArrayList<User>();
  for(User user: userRepository) {
    if (user.getAge() > age)
      users.add(user);
  }
  return users;
}
```

```java
public List<User> getUsersWithAgeOlderThan(int age) {
  return userRepository.stream().filter(user -> user.getAge()>age).toList();
}
```

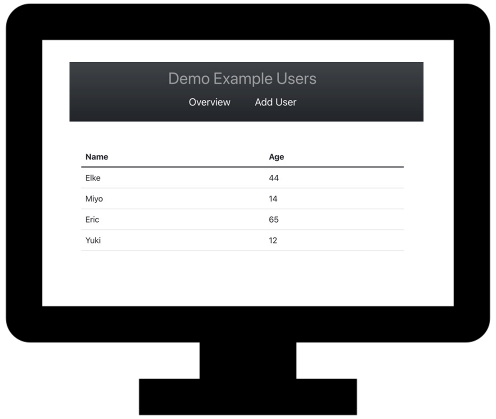
- JavaScript

```
userRepository.
  filter((user) => user.age>age)
```

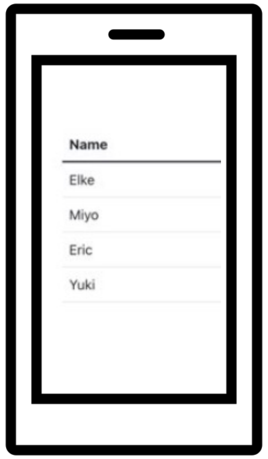
- Java

```
userRepository.
  stream().
    filter(user -> user.getAge()>age).
      toList();
```

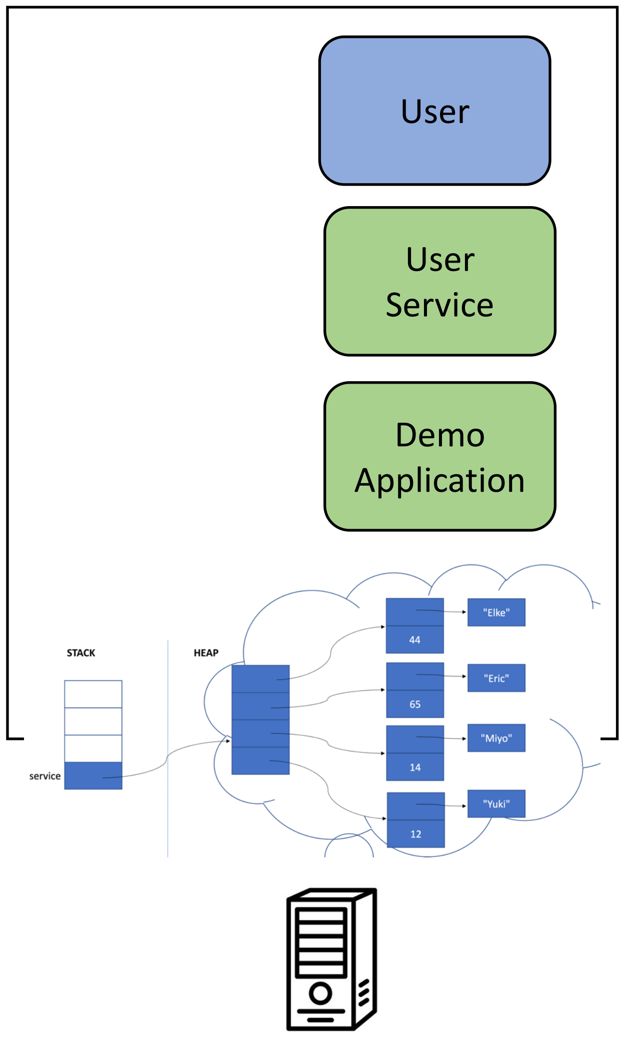
END GOAL - STEP 2



Name	Age
Elke	44
Miyo	14
Eric	65
Yuki	12

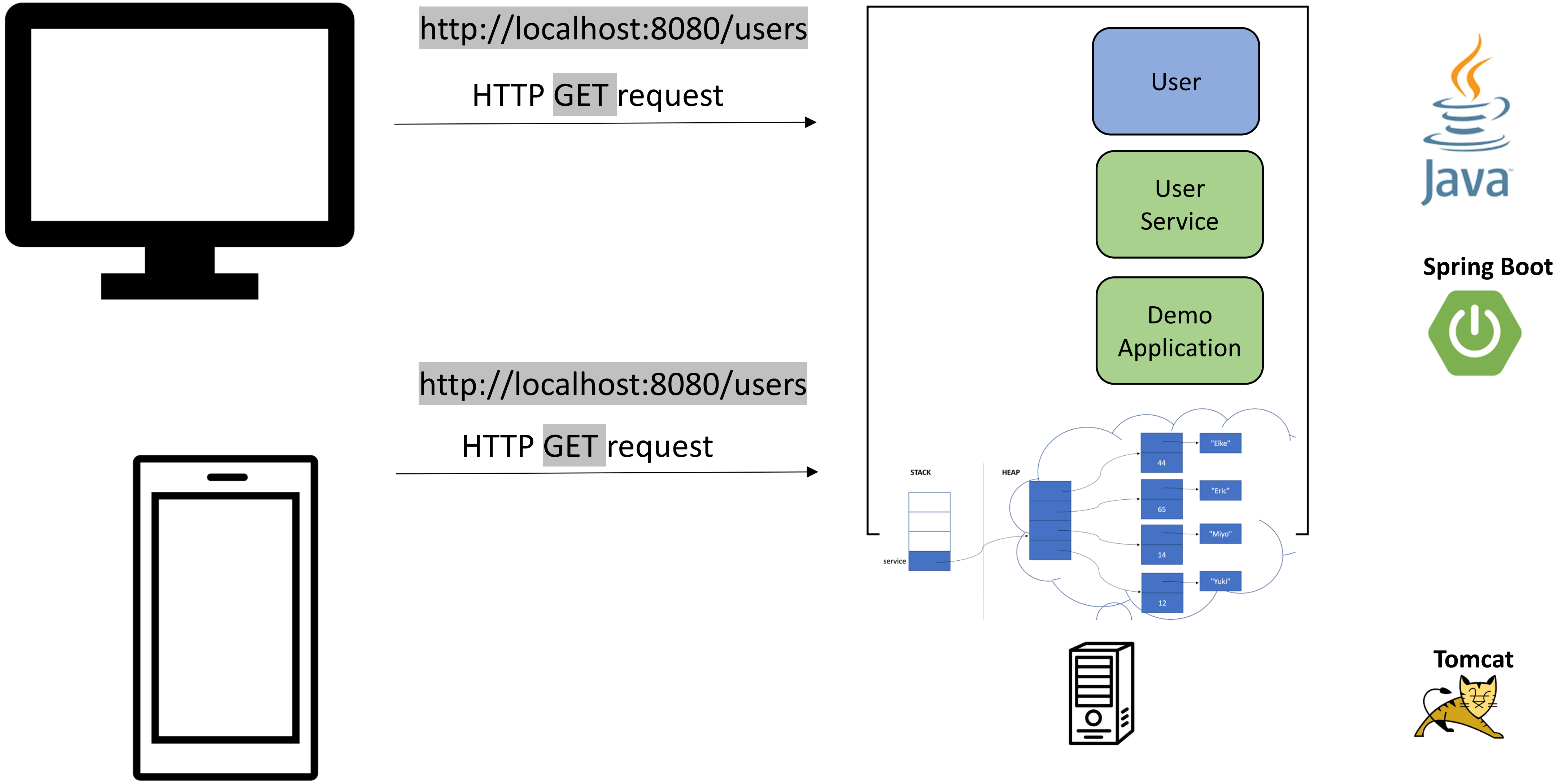


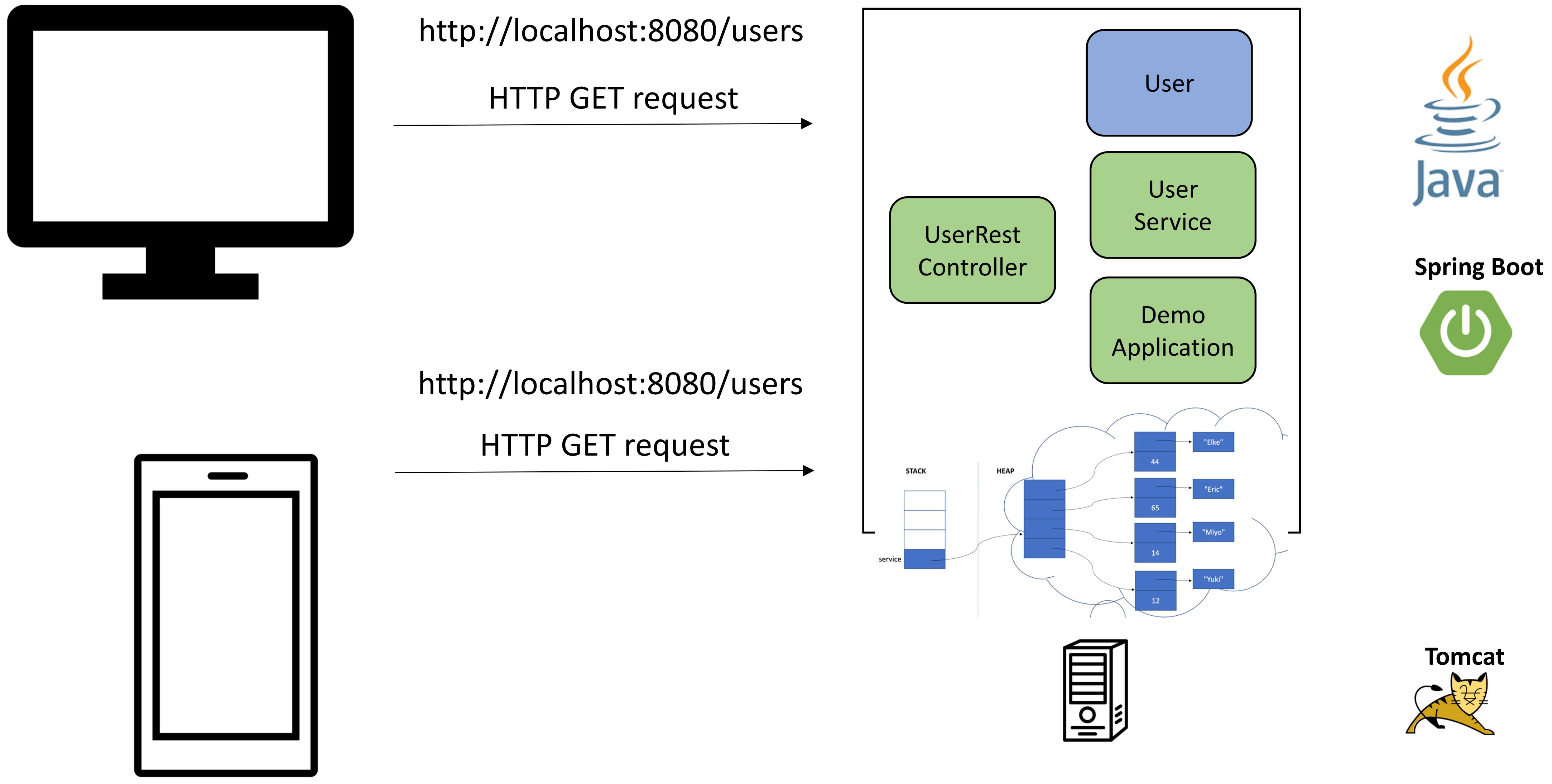
Name
Elke
Miyo
Eric
Yuki

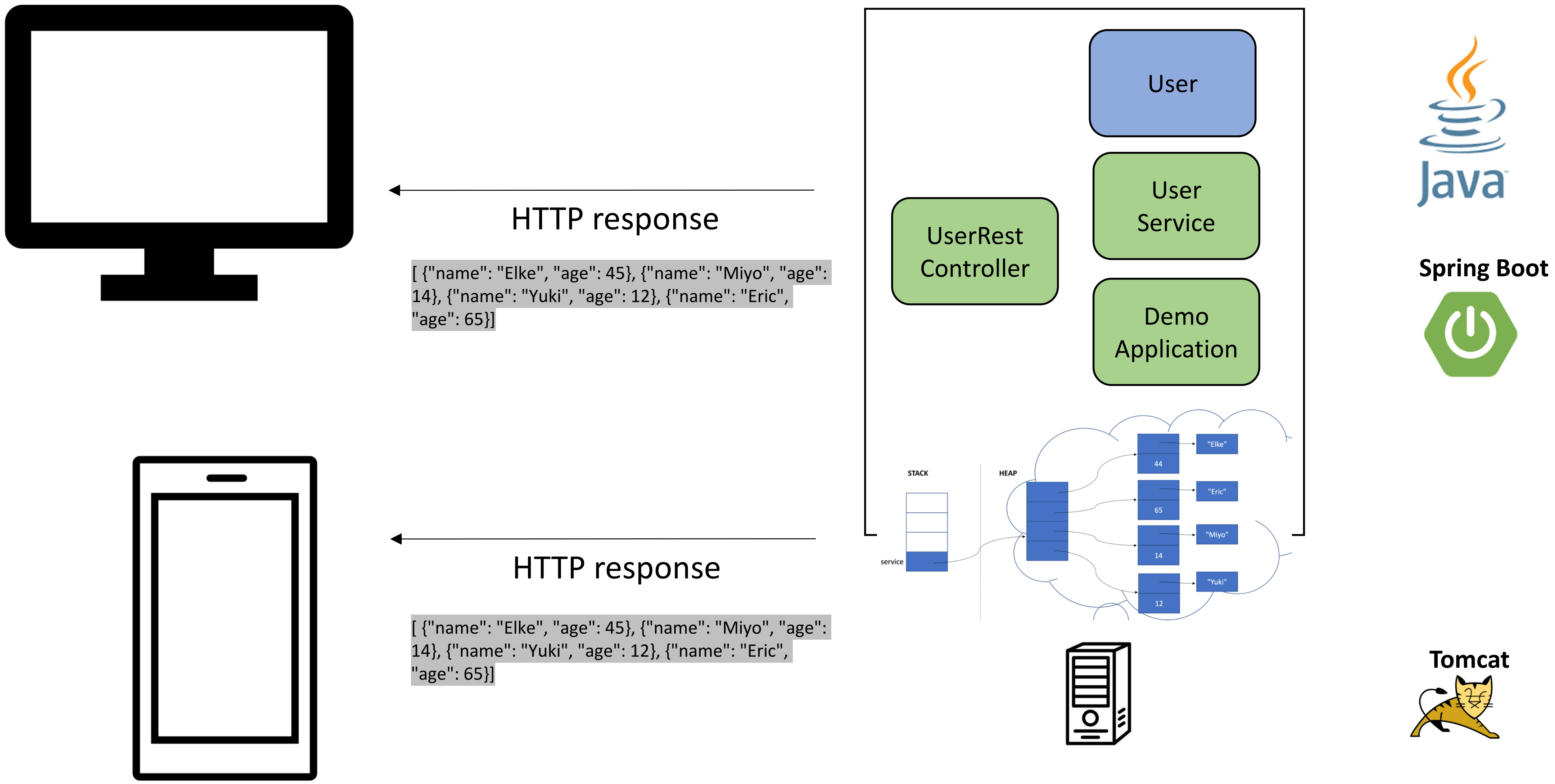


Spring Boot

Tomcat

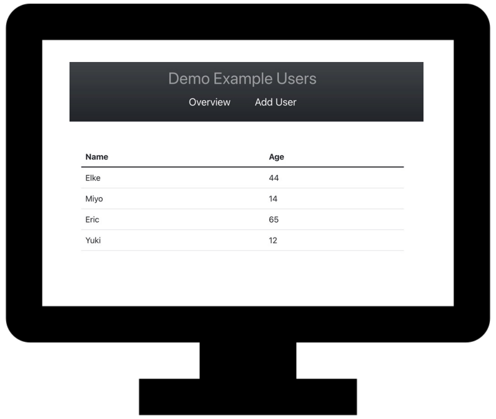

http://localhost:8080/users
HTTP GET request
http://localhost:8080/users
HTTP GET request
User
User Service
Demo Application
STACK
HEAP
service
44
65
14
12
"Elke"
"Eric"
"Miyo"
"Yuki"
Java
Spring Boot
Tomcat


http://localhost:8080/users
HTTP GET request
http://localhost:8080/users
HTTP GET request
User
User Service
UserRest Controller
Demo Application
STACK
HEAP
service
44
"Elke"
65
"Eric"
14
"Miyo"
12
"Yuki"
Java
Spring Boot
Tomcat



```
[ {"name": "Elke", "age": 45}, {"name": "Miyo", "age": 14}, {"name": "Yuki", "age": 12}, {"name": "Eric", "age": 65}]
```

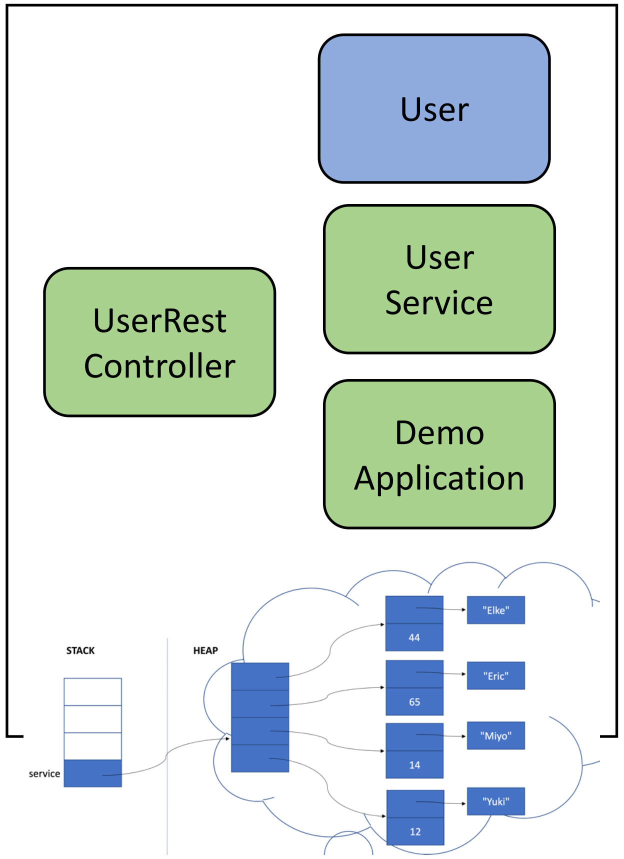
```
[ {"name": "Elke", "age": 45}, {"name": "Miyo", "age": 14}, {"name": "Yuki", "age": 12}, {"name": "Eric", "age": 65}]
```


Demo Example Users

Overview Add User

Name	Age
Elke	44
Miyo	14
Eric	65
Yuki	12

Name
Elke
Miyo
Eric
Yuki



Spring Boot

Tomcat

REST Controller

deze stuurt de date in gevraagde paket vorm op naar de GET-requests

```
public class UserRestController {

  private UserService userService;

  // TODO

}
```

@RestController

- to create a RESTful web service in a simplified manner
- indicates that the data returned by each method will be written straight into the response body instead of rendering a template
  - every request handling method of the controller class automatically serializes return objects into response body

```
@RestController
@RequestMapping("/users")
public class UserRestController {

  @Autowired
  private UserService userService;

  @GetMapping
  public List<User> getAllUsers() {
    return userService.getAllUsers();
  }
}
```

RESTful web service

- is a lightweight, maintainable, and scalable service that is built on the REST architecture
- Restful Web Service, expose API from your application in a secure, uniform, stateless manner to the calling client. The calling client can perform predefined operations using the Restful service.
- REST stands for Representational State Transfer = Overdracht van representatieve staat
- result is a REST API = REST Application Programming Interface

@RequestMapping

- is used to map web requests to Spring Controller methods
  - indicates that all URIs with .../users/... in the path will be executed by this rest controller

```
@RestController
@RequestMapping("/users")
public class UserRestController {

  @Autowired
  private UserService userService;

  @GetMapping
  public List<User> getAllUsers() {
    return userService.getAllUsers();
  }
}
```

@Autowired

- indicates that the marked dependency is injected automatically by Spring

```
@RestController
@RequestMapping("/users")
public class UserRestController {

  @Autowired
  private UserService userService;

  @GetMapping
  public List<User> getAllUsers() {
    return userService.getAllUsers();
  }
}
```

Dependency Injection (DI)

dependency injection is a design pattern in which an object or function receives other objects or functions that it depends on

- is a fundamental aspect of the Spring framework, through which the Spring container “injects” objects into other objects or “dependencies”
- Simply put, this allows for loose coupling of components and moves the responsibility of managing components onto the container.

Inversion of Control (IoC)

- is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework
- can be achieved through various mechanisms such as
  - Strategy pattern
  - Factory pattern
  - Dependency Injection
    - the Spring container "injects" objects into other objects or "dependencies"

@GetMapping

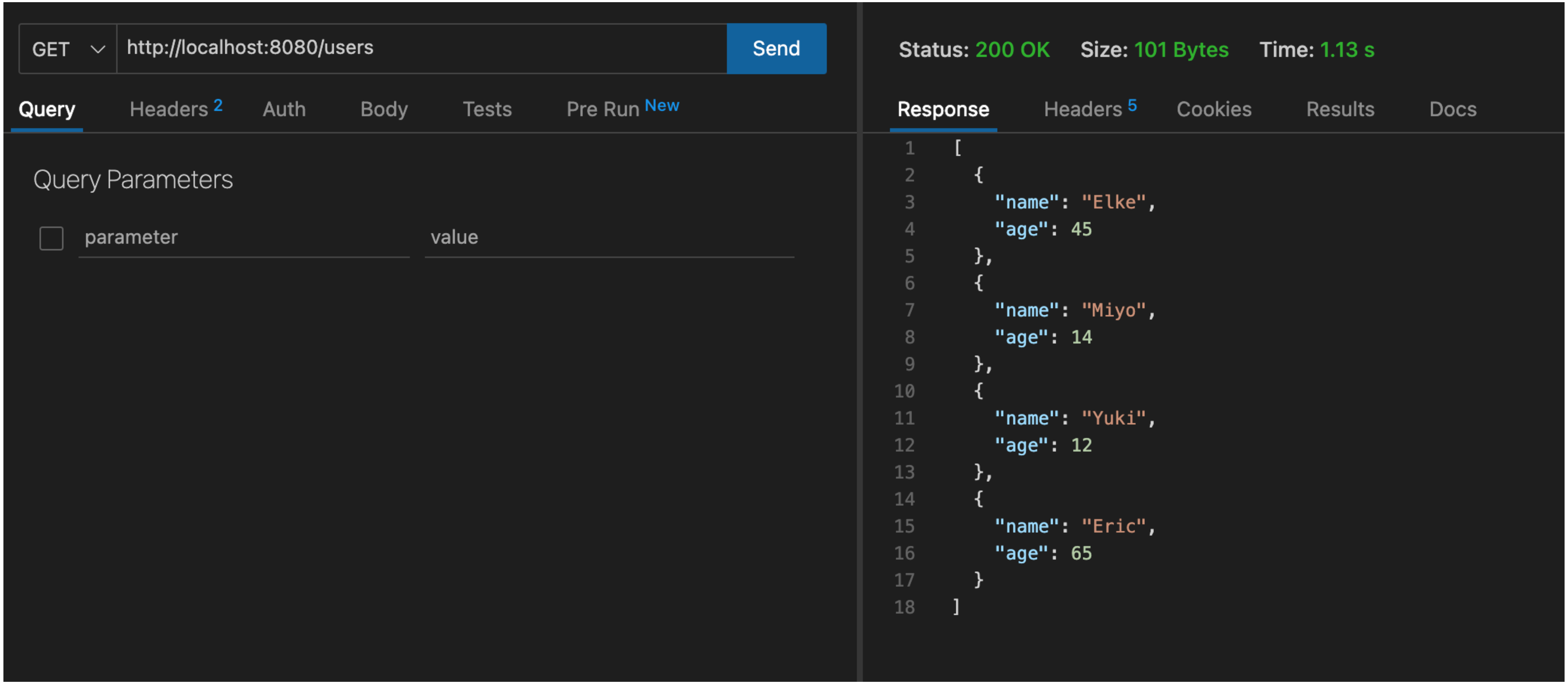
- annotation to handle proper incoming HTTP GET methods with URI
  - GET request with URI localhost:8080/users/

```
@RestController
@RequestMapping("/users")
public class UserRestController {

  @Autowired
  private UserService userService;

  @GetMapping
  public List<User> getAllUsers() {
    return userService.getAllUsers();
  }
}
```

Manually Testing your REST API

- Therefor you need
  - a good tool
    - Thunder Client for VS Code
      - = hand-crafted lightweight Rest Client for Testing APIs
  - your critical brain :-)
    - Thinking about the different outcomes to test

- Later we will automate the testing of your REST API ...



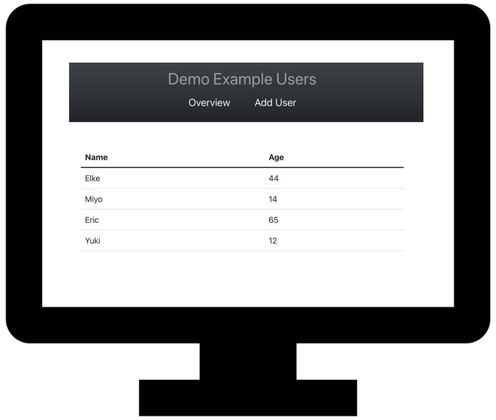
```
http://localhost:8080/users
```

Status: 200 OK Size: 101 Bytes Time: 1.13 s

```
[
  {
    "name": "Elke",
    "age": 45
  },
  {
    "name": "Miyo",
    "age": 14
  },
  {
    "name": "Yuki",
    "age": 12
  },
  {
    "name": "Eric",
    "age": 65
  }
]
```

CLIENT

SERVER



Demo Example Users

Name	Age
Elke	44
Miyo	14
Eric	65
Yuki	12

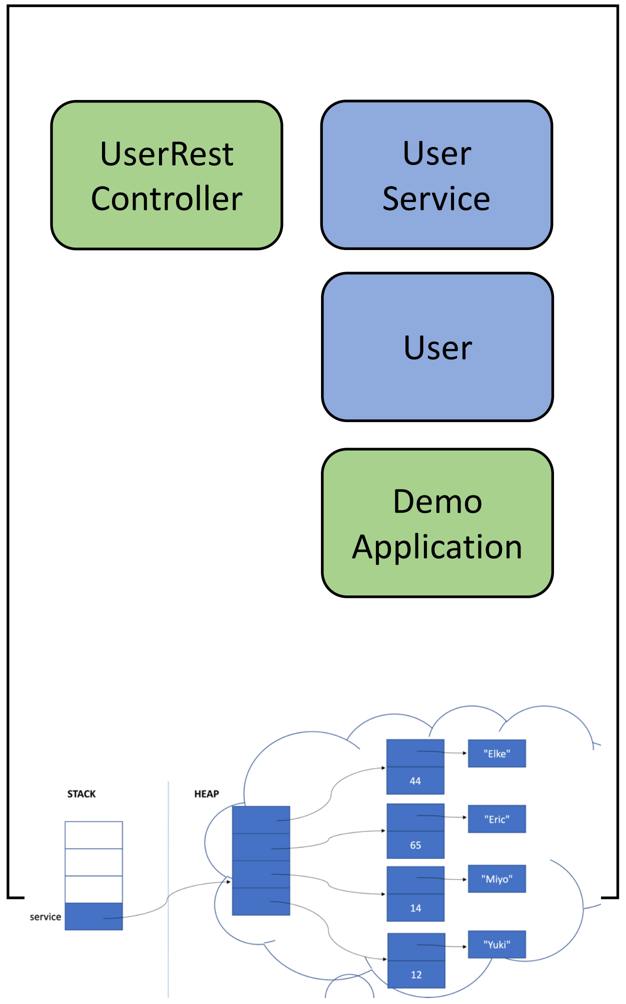
http://localhost:8080/users

HTTP GET request

HTTP response

200 OK
in the body

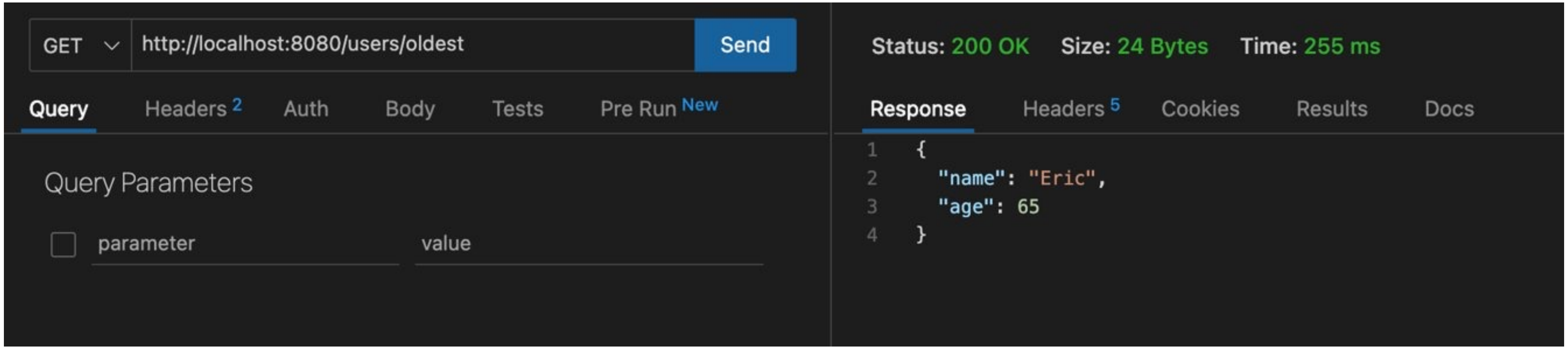
```
[
 {
   "name": "Elke",
   "age": 45
 },
 {
   "name": "Miyo",
   "age": 14
 },
 {
   "name": "Yuki",
   "age": 12
 },
 {
   "name": "Eric",
   "age": 65
 }
]
```



@GetMapping continued ...

- URI needs to be unique
  - we have 2 GET requests
    - one with URI localhost:8080/users/
      - returning all users
    - another with URI localhost:8080/users/oldest
      - returning only the oldest user

```
@RestController
@RequestMapping("/users")
public class UserRestController {

  @Autowired
  private UserService userService;

  @GetMapping
  public List<User> getAllUsers() {
    return userService.getAllUsers();
  }

  @GetMapping("/oldest")
  public User getOldestUser() {
    return userService.getOldestUser();
  }

}
```



http://localhost:8080/users/oldest

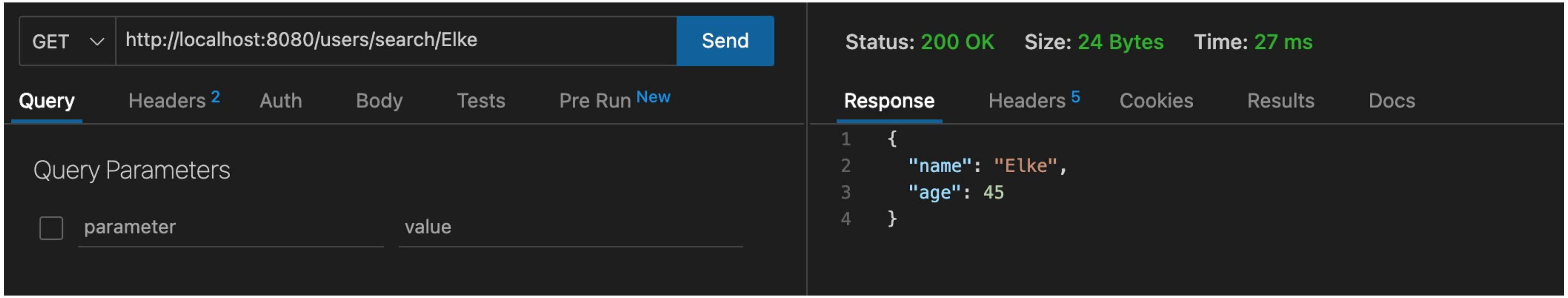
Status: 200 OK Size: 24 Bytes Time: 255 ms

```
{
  "name": "Eric",
  "age": 65
}
```

@PathVariable

- can be used to handle template variables in the request URI mapping, and set them as method parameters
  - URI localhost:8080/users/search/Elke
    - returning only the information in JSON format of the user Elke

```
@RestController
@RequestMapping("/users")
public class UserRestController {


  …


 @GetMapping("/search/{name}")
 public User searchUserWithName
   (@PathVariable("name") String name) {
     return userService.getUserWithName(name);
 }


}
```



http://localhost:8080/users/search/Elke

Status: 200 OK Size: 24 Bytes Time: 27 ms

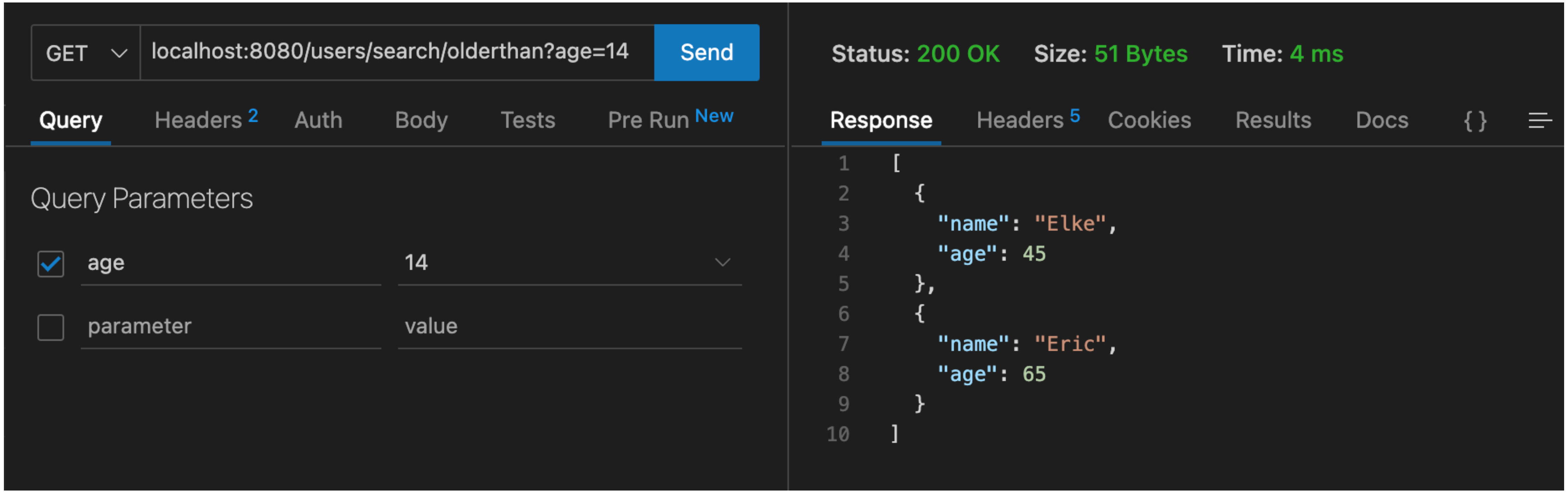
Query Parameters

```
{
  "name": "Elke",
  "age": 45
}
```

@RequestParam

- to extract query parameters
  - URI localhost:8080/users/search/olderthan?age=14
    - returning only the users with age 14 in JSON format

```
@RestController
@RequestMapping("/users")
public class UserRestController {

  …

 @GetMapping("/search/olderthan")
 public List<User> searchUsersWithAgeOlderThan
  (@RequestParam("age") int age) {
    return
    userService.getUsersWithAgeOlderThan(age);
}

}
```



localhost:8080/users/search/olderthan?age=14

Query Parameters

✓	age	14
	parameter	value

Status: 200 OK Size: 51 Bytes Time: 4 ms

```
[
  {
    "name": "Elke",
    "age": 45
  },
  {
    "name": "Eric",
    "age": 65
  }
]
```

Front-End using your Back-End

- Important is to use the agreed URIs in the REST API because the agreed URIs are the glue between the back-end and the front-end :-)

@CrossOrigin

- includes headers for Cross-Origin Resource Sharing (CORS) in the response
- placing it on class level enables CORS on all handler methods of this class

```
@CrossOrigin(origins = "http://127.0.0.1:3000"
)
@RestController
@RequestMapping("/users")
public class UserRestController {

…

}
```

Front-End

```
const fetchAndRenderUsers = async () => {
  ...
  users.length = 0
  const response = await fetch("http://localhost:8080/users")
  const result = await response.json()
  users.push(...result)
}
```

- call to REST API
  - URI localhost:8080/users/
    - returning json with data of all users in back-end
- uses the returned json to render the data on the HTML in the browser of the client
- using fetch JS function

References

- https://spring.io/guides/gs/rest-service/

UserService = maakt functies zoals "GetOldest", "GetUsers", "leeftijd boven ..." (voor deze heb je REQUESTPARAM nodig)

UserRestController = dit is de communicatie tussen de frond-end & back-end