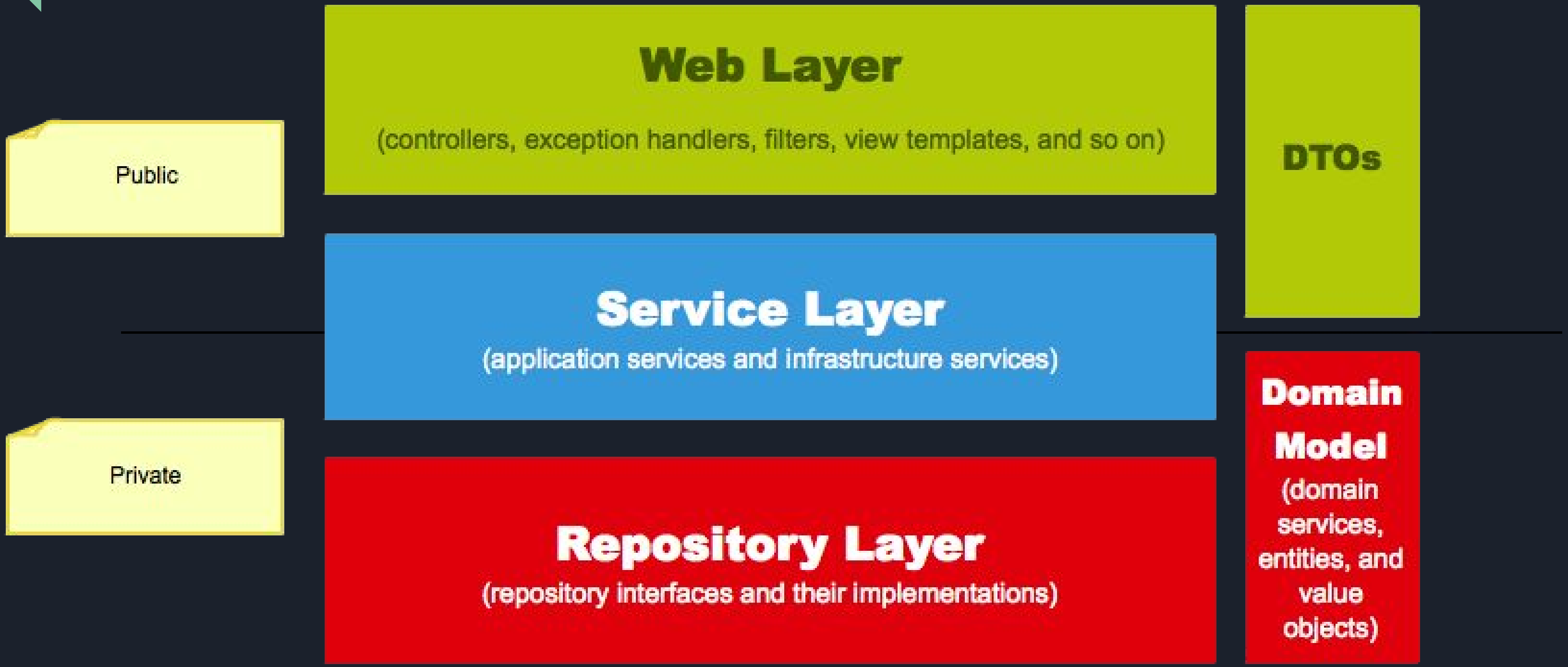
Spring Boot Workshop

Merve Sahin

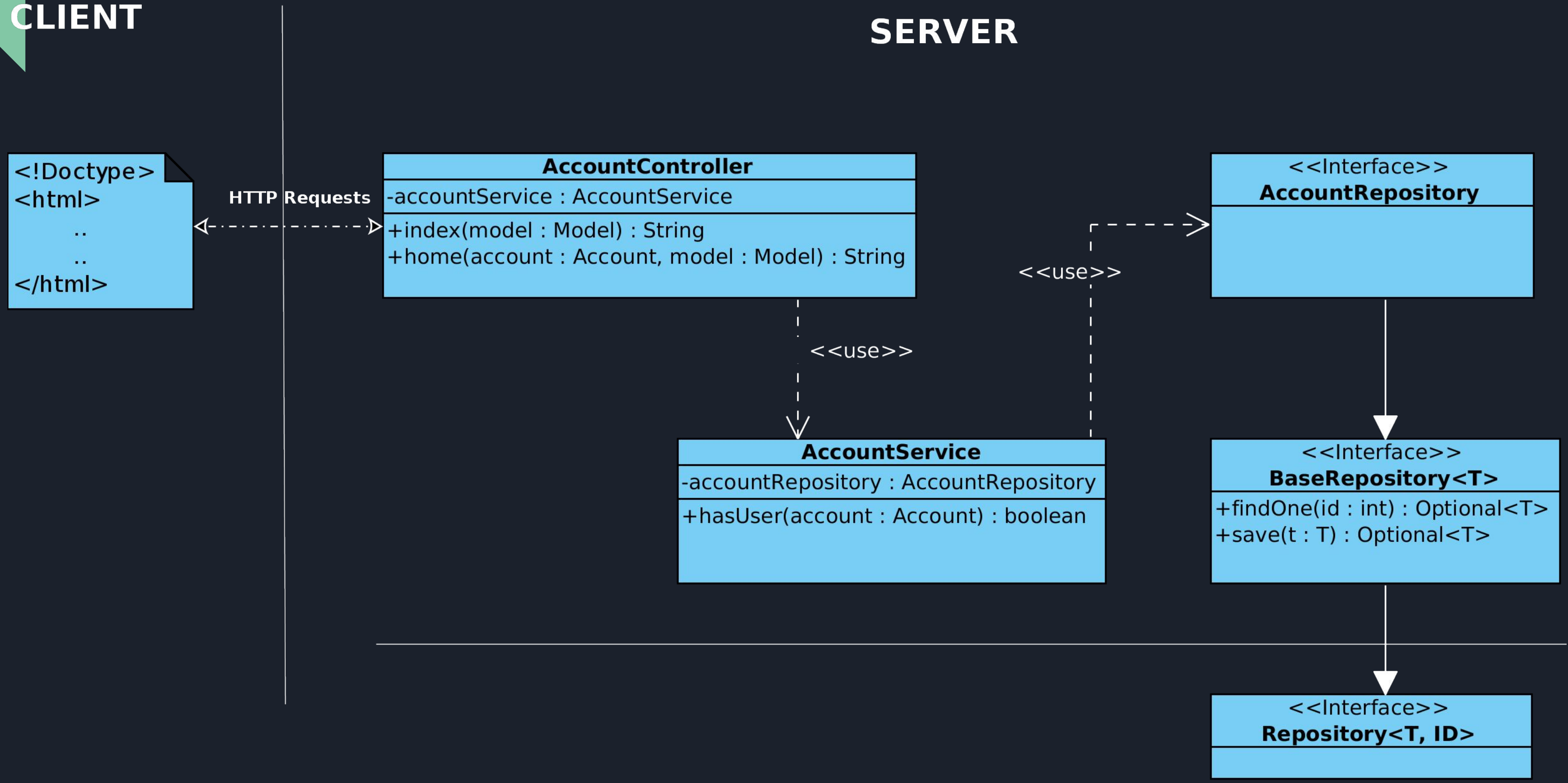
Content

- Spring Boot Structure
- Client-Server Interaction
- Dependency Injection
- Object Relational Mapping with JPA
- Thymeleaf Template Engine
- Spring Security
- Testing

Spring Boot Structure


Public
Private
Web Layer
(controllers, exception handlers, filters, view templates, and so on)
Service Layer
(application services and infrastructure services)
Repository Layer
(repository interfaces and their implementations)
DTOs
Domain Model
(domain services, entities, and value objects)

Client-Server Interaction


CLIENT
SERVER
<!Doctype>
<html>
..
..
</html>
HTTP Requests
AccountController
-accountService : AccountService
+index(model : Model) : String
+home(account : Account, model : Model) : String
<<use>>
<<use>>
<<Interface>>
AccountRepository
AccountService
-accountRepository : AccountRepository
+hasUser(account : Account) : boolean
<<Interface>>
BaseRepository<T>
+findOne(id : int) : Optional<T>
+save(t : T) : Optional<T>
<<Interface>>
Repository<T, ID>

Dependency Injection

- @Autowired injects instances of Services, Repositories and many other Beans
- Beans need to be annotated for spring to be able to inject them:
  - Controller -> @Controller OR @RestController
  - Service -> @Service
  - Repository -> @Repository

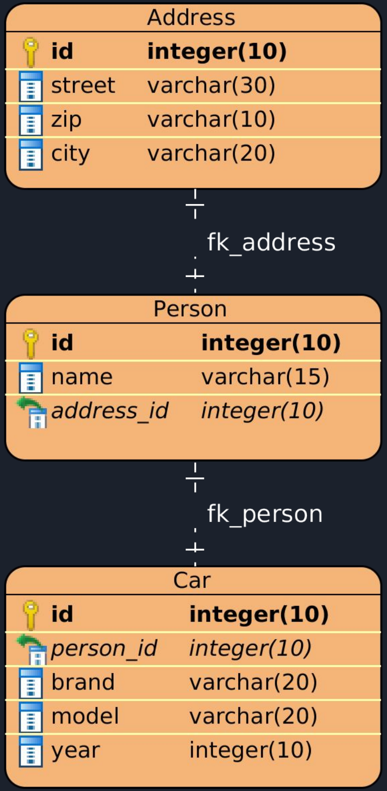
```
@Service
public class AccountService{
    private AcccountRepository accountRepository;

    @Autowired
    public AccountService(AccountRepository ar){
        this.accountRepository = ar;
    }
}
```

5 Steps to creating a Web-Application

1.	Mapping Relational Tables to Java Classes	->	*Entity*
2.	Defining Repositories and Methods for querying	->	*Repository*
3.	Implementing Services to process data provided by Repositories	->	*Service*
4.	Creating a Rest-Endpoint for consuming & producing data	->	*Controller*
5.	Implementing a Client	->	*HTML, CSS, JS*

Object Relational Mapping with JPA



```java
@Entity
@Table(name = "person")
public class Person{

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name="name")
    private String name;

    @JoinColumn(name="address_id", referencedColumnName="address")
    @ManyToOne
    private Address address;

    @OneToMany(mappedBy="personId")
    private Set<Car> cars;
    ...
}
```

Querying in Repositories

- Queries are created using method signatures
- Query structure: [*operation*]:[*option1*]:[*where*]:[*parameter*]:[*option2*]:[*operand*]: ...

Operation	=	find, delete, count, exists
Option1	=	distinct, ..
Where	=	by
Parameter	=	column name
Option2	=	equalsIgnoreCase, startsWithIgnoreCase, ..
Operand	=	and, or

Example:

```
public Optional<User> findByNameEqualsIgnoreCaseAndAge(String name, int age);
```

See Documentation: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

Mapping Web Requests To Controllers

Javascript - Client

```
var user={
    ..
};
$.ajax({
    url : "/user/save",
    type: "POST",
    contentType: "application/json",
    dataType : 'json',
    data: JSON.stringify(user)
});
```

Controller - Server

```
@PostMapping(value="/save",
     consumes ={MediaType.APPLICATION_JSON_VALUE})

@ResponseStatus(value = HttpStatus.OK)
public void save(@RequestBody StudentDTO student ){
    ..
}
```

Mapping Web Requests To Controllers 2

Javascript - Client

```
$.ajax({
    url: "/user/id/1",
    type: "GET",
    success: function(userDto){
        ..
    }
});
```

Controller - Server

```
@Controller
@RequestMapping("/user")
public class LoginController{

    @GetMapping("/id/{id}")
    @ResponseBody
    public UserDTO getUser(@PathVariable("id") int id){
        ..
        return userDTO;
    }
}
```

Thymeleaf Template Engine

- Binding models into HTML using Thymeleaf:

edit.html

```html
<form method="post" action="#" th:action="@{/user/add}" th:object="${user}">
    <p>Name:
        <input type="text" th:field="*{name}">
        </input>
    </p>
    ...
    <p> <input type="submit" value="submit"></input> </p>
</form>
```

Controller.java

```java
@RequestMapping("/edit")
public String index(Model model){
    model.add("user", new UserDTO());
    return "edit";
}

@PostMapping("/add")
public String addUser(@ModelAttribute("user") UserDTO user){
    ..
}
```

Thymeleaf Template Engine 2

- Iterating through arrays using ‘th:each’:

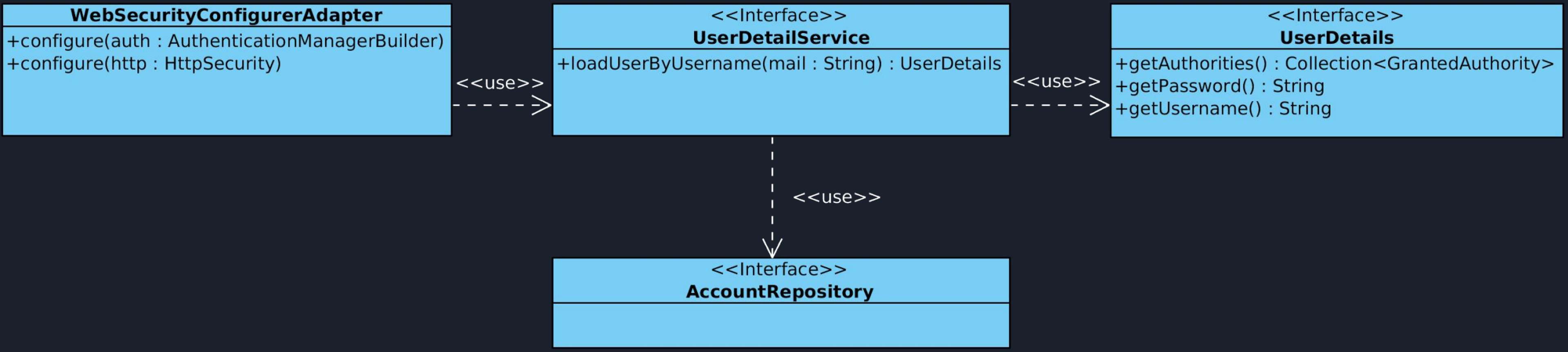
```
<table>
    <tr>
        <th>Name</th>
    </tr>
    <tr th:each="user : ${users}" >
        <td th:text="${user.name}"></td>
    </tr>
</table>
```

TASK 1

TASK 2

- Implement Controller, Service and Repository for Student
- Define a query for retrieving courses of the students current semester
- Use Thymeleaf to display a list of students

Spring Security


WebSecurityConfigurerAdapter
+configure(auth : AuthenticationManagerBuilder)
+configure(http : HttpSecurity)
<<use>>
<<Interface>>
UserDetailService
+loadUserByUsername(mail : String) : UserDetails
<<use>>
<<Interface>>
UserDetails
+getAuthorities() : Collection<GrantedAuthority>
+getPassword() : String
+getUsername() : String
<<use>>
<<Interface>>
AccountRepository

Spring Security 2

- Extending *WebSecurityConfigurerAdapter* to configure Security:

```java
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfiguration extends WebSecurityConfigurerAdapter{
    @Autowired
    private UserDetailsService service;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(service);
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("**/secured/**").authenticated()
                .anyRequest().permitAll()
                .and().formLogin().permitAll();
    }
}
```

Spring Security 3

- Accessing Methods with certain permissions:

```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/edit/user/{id}")
public String edit(@PathVariable("id")int id, Model model){
    ..
    return "edit";
}
```

- Granting permission to several roles:

```
@PreAuthorize("hasAnyRole('ADMIN', 'USER')")
```

TASK 3

- Implement a custom UserDetailsService and UserDetail
- Restrict access to methods with pre authorization

Testing with MockMvc

- Initializing a Test Class:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
@SpringBootTest
public class AccountControllerTest {

    @Autowired
    private WebApplicationContext context;
    private MockMvc mvc;

    @Before
    public void setUp() {
        mvc = MockMvcBuilders
                .webAppContextSetup(context)
                .apply(SecurityMockMvcConfigurers.springSecurity())
                .build();
    }
}
```

Testing With MockMvc 2

- Creating integration tests:

```
@Test
public void testIndex() throws Exception {
    mvc.perform(get("/user/3").with(user("user").password("pass").roles("ADMIN","USER")))

     .andExpect(status().isOk())                                      HTTP Status
     .andExpect(content().contentType(CONTENT_TYPE_HTML))             Content Type of return
     .andExpect(view().name("index"))                                 Name of HTML Template
     .andExpect(model().attribute("account",
            Matchers.hasProperty("mail", IsNull.nullValue())));       Attributes in Model
}
```

See Documentation:
https://docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-framework

TASK 4