Hugbúnaðarverkefni 1 / Software Project 1

5. Web Applications

HBV501G – Fall 2018

Matthias Book

HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ
IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI- OG TÖLVUNARFRÆÐIDEILD

In-Class Quiz Prep

- Please prepare a scrap of paper with the following information:

- ID: _____@hi.is Date: _____
- a) _____ e) _____
- b) _____ f) _____
- c) _____ g) _____
- d) _____ h) _____

- During class, I'll show you questions that you can answer very briefly
  - No elaboration necessary
- Hand in your scrap at the end of class

- All questions in a quiz weigh same
- All quizzes (ca. 10 throughout semester) have the same weight
  - Your worst 2 quizzes will be disregarded
- Overall quiz grade counts as optional question worth 7.5% on final exam

Quiz #3 Solution: Testing Use Cases' Validity

- **Boss test**
  - Would your boss be happy if you told him you performed this use case all the time?
  - Checks whether the use case describes actions that **create value**.
- **Size test**
  - Most use cases are on a **time scale** of a few minutes to an hour.
    - Neither something done in one small step, nor something taking several sessions or days to complete is likely a valid use case.
  - Make sure it's not just one step of a more complex use case.
    - If the fully dressed format is less than one **page** long, it's probably not a use case.
- **EBP test**
  - Does the use case represent an elementary business process (EBP)?
    - A task performed by one person in one place at one time,
    - in response to a business event,
    - which adds measurable business value
    - and leaves the data in a consistent state.
- **Are the following valid use cases?**
  a) Buy an apartment (n)
  b) Get a refund for a returned product (y)
  c) Log in to a system (y/n)
  d) Check a kennitala's syntactic validity (n)
  e) Subscribe to an e-mail newsletter (y)

Recap: Team Assignment 2

- By **Sun 30 Sep**, submit in Ugla:
  - An initial **Use Case document** (text-only, no UML diagrams)
    - Describe the two most complex requirements mentioned in the Vision & Scope document
      - Use *fully dressed use case* template on next slide
      - Sections 1, 4, 6, 7, 8, 9 and 13 are mandatory for assignment
- On **Thu 4 Oct**, present and **explain** your document to your tutor:
  - How did you come up with your use cases and scenarios?
  - Why are these the most important ones?
  - What further information might have to be considered before implementation?
- **Grading criteria**
  - 2 main use cases in fully-dressed format (50% each)
    - Precise and plausible information in Sect. 1, 4, 6, 7, 13 (10%)
    - Precise formulation of main scenario in Sect. 8 (20%)
    - Precise formulation of alternative scenarios in Sect. 9 (20%)

Recap: Fully Dressed Use Case Format Template

Assignment 2

1. **Use case name**
   - What are these scenarios about? (Start with a verb.)
2. **Scope**
   - What is the system under design?
3. **Level**
   - Is this use case describing a user's goal (top level) or a sub-function of some other use case?
4. **Primary actor**
   - Who calls on the system to deliver its services?
5. **Stakeholders and interests**
   - Who cares about this use case, and what do they want?
6. **Preconditions**
   - What must be true at the beginning of the scenario (and worth telling the reader)?
7. **Success guarantee**
   - What must be true on successful completion (and worth telling the reader)?
8. **Main success scenario**
   - What is the typical, unconditional "happy path" scenario of success?
9. **Extensions / alternate scenarios**
   - What are alternate scenarios of success or failure?
10. **Special requirements**
    - What are related non-functional requirements?
11. **Technology and data variations list**
    - What varying input/output methods and data formats should we be aware of?
12. **Frequency of occurrence**
    - How often does this use case occur? (Influences investigation, timing, priority, testing…)
13. **Miscellaneous / open issues**
    - What open issues are there?

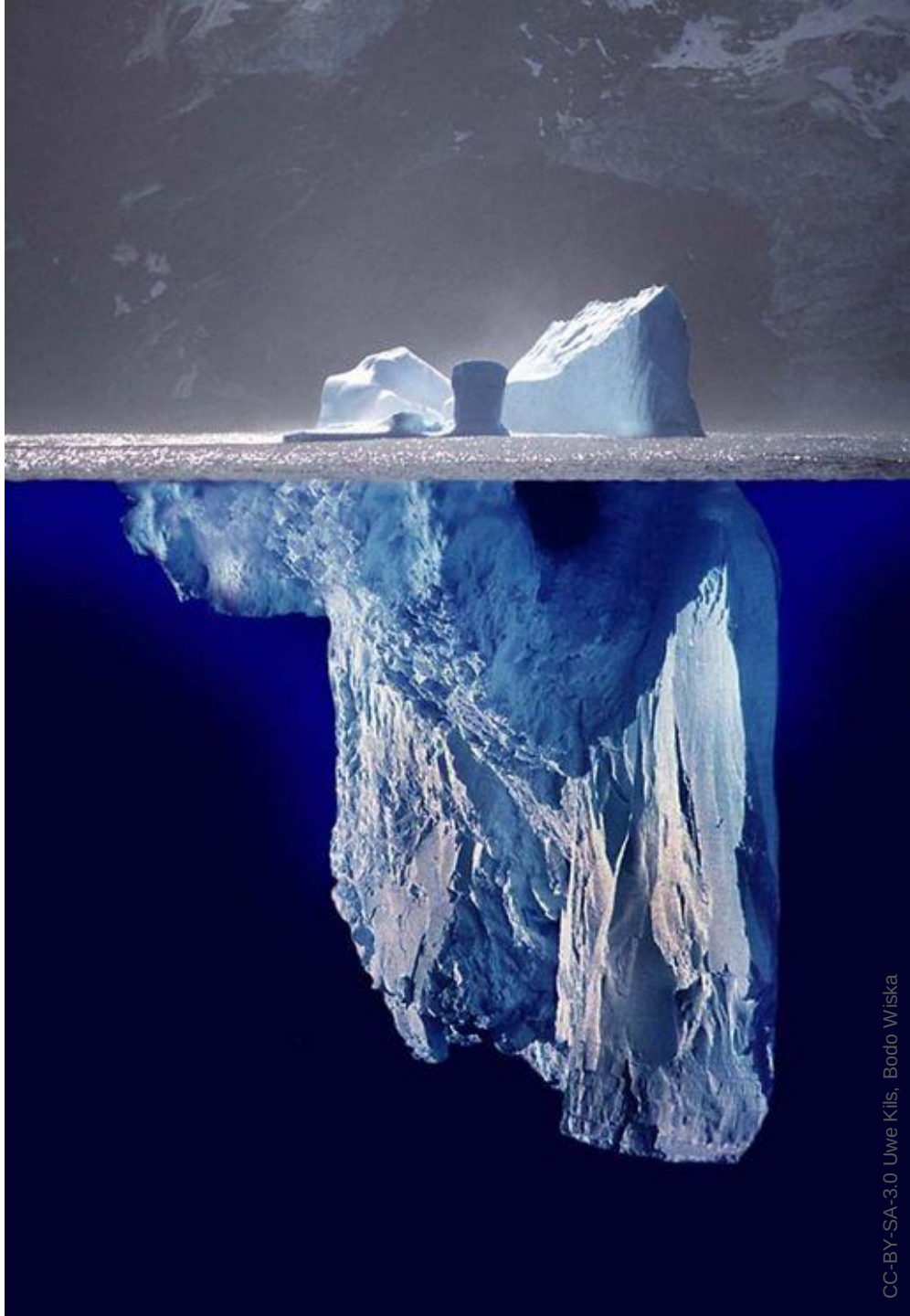
Web Application Development
with Spring Web MVC

see also:

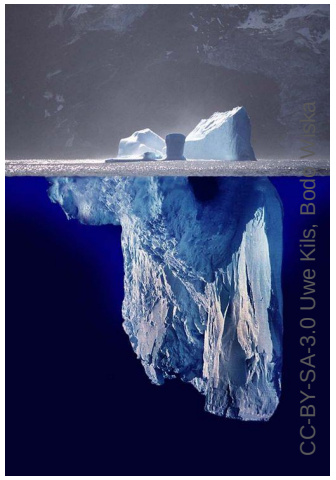
- https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#spring-web
- http://spring.io/guides/gs/spring-boot/
- http://spring.io/guides/gs/serving-web-content/

Motivation: Application Layers
(Simplified View)

- User interface logic
- Business logic



Model-View Separation Principle

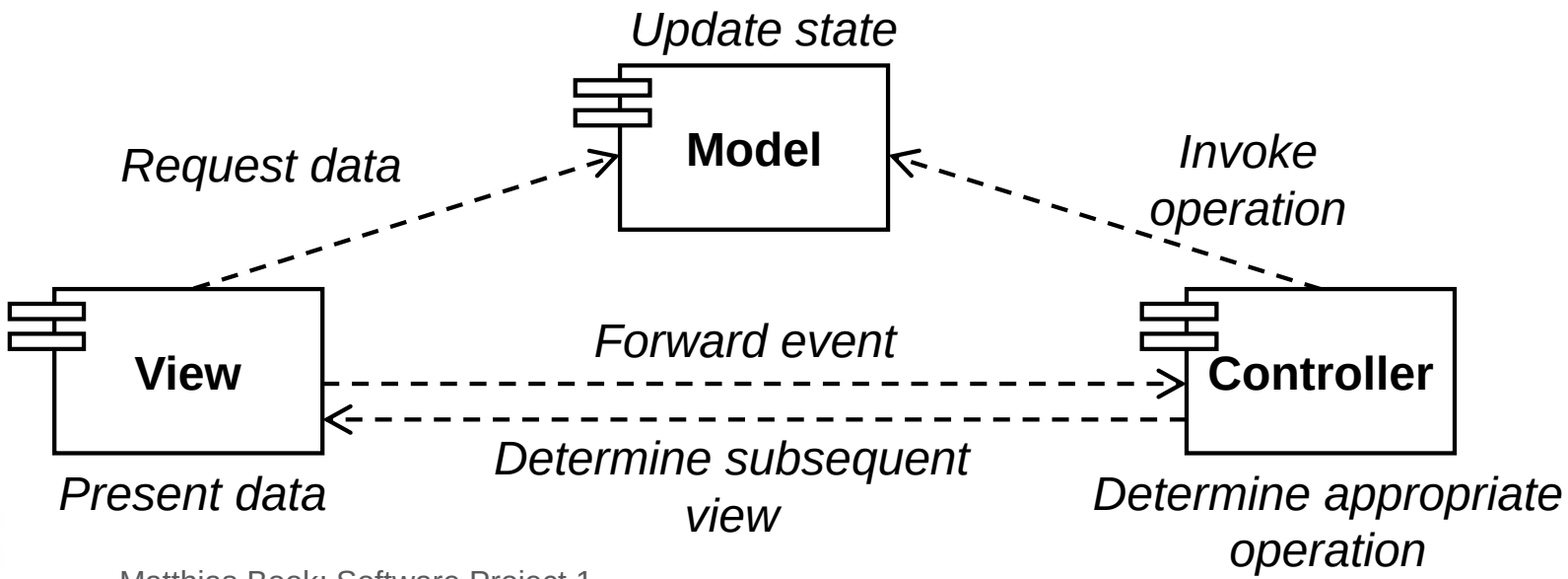


- The interface between user interface (UI) logic and business logic can easily become messy:
  - Both sides deal with very similar domain concepts, but represent them differently
  - Both sides need to be aware of business processes and rules, but to different degrees
  - Both sides could do the job of the other side, although not always as cleanly
  - Both sides could be decoupled from the other, but to which degree is it reasonable?
  - Both sides' dialog and control flows depend closely on each other
  - Without a clean separation, maintainability can become a nightmare

- Solution: Strict distinction of
  - **Model** components – comprising the business objects of the application domain
  - **View** components – comprising the user interface elements
  - **Controller** components – comprising the application logic that is in charge of the control flow

Model-View-Controller Pattern

- **Model:** Application state (data and operations working on them); usually subdivided again into the working data (in memory) and the persistence layer
- **View:** View(s) of a state; usually the user interface, but possibly other access services as well. *The view does not change the model!*
- **Controller:** Input interface; invokes particular operations on the model, based on events coming in from the view



Model-View-Controller Pattern

- Strict distinction of
  - **Model** components – comprising the business objects of the application domain
  - **View** components – comprising the user interface elements
  - **Controller** components – comprising the application logic that is in charge of the control flow

- Implications and responsibilities:
  - A **business object** (e.g. Contract, Product, Customer, Sale…) will not call or rely on a user interface object (e.g. a window, web page, input field…).
    - It will only provide information to user interface objects when requested
  - A **user interface** (UI) object will not implement application logic (e.g. calculating tax).
    - It will only initialize UI objects with model data, receive UI events (e.g. mouse clicks), and delegate requests for functionality to appropriate controllers.
  - **Controller objects** receive messages from the UI, call the appropriate business functions, update the data model, and return control back to the user interface for the next interaction.

Web Applications

- In a web application, the architecture encompasses two communicating actors:
  - Client – usually, a web browser running on a desktop or mobile device
  - Server – a central host upon which the clients depend to provide data and/or computation

Common implementation options:

- **Client-side MVC**
  - Application's views are created on the client, dialog flow is controlled by the client
  - Server provides data and services (e.g. through RESTful services)

- **Server-side MVC**
  - Application's views are created on the server, dialog flow is controlled by the server
  - Client just displays the views and submits user inputs to server

HTTP Request/Response Cycle

- User clicks on link
- User submits form
- Script requests data

- Request for particular resource (URI), incl.
- session identifier
- optional parameters

- Establishes session context
- Decides how to handle request
- Executes app logic
- Constructs response

HTTP Request

HTTP Response

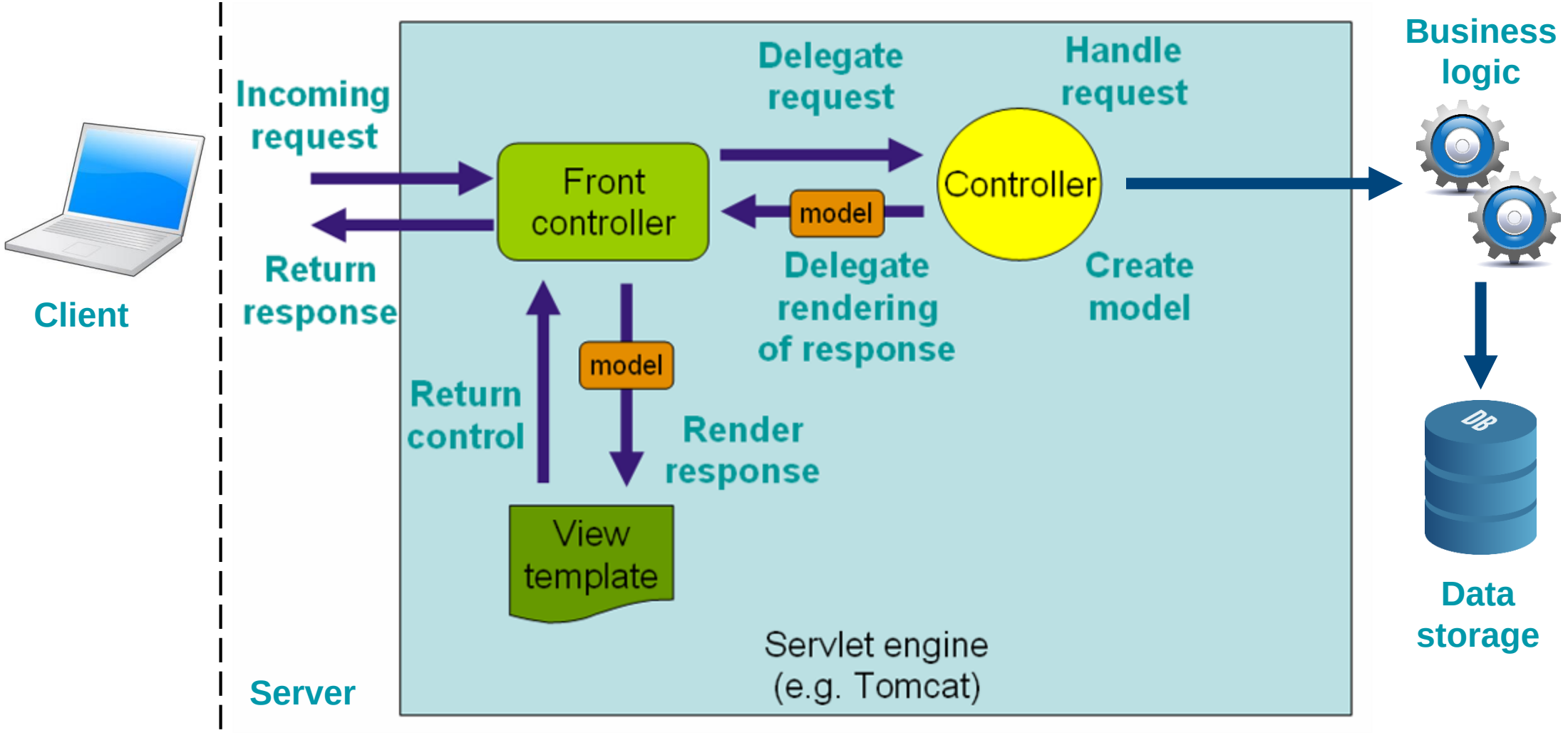
Client
(typ.: web browser)

Server
(incl. application server, database server, etc.)

- Displays web page or
- Updates view with data

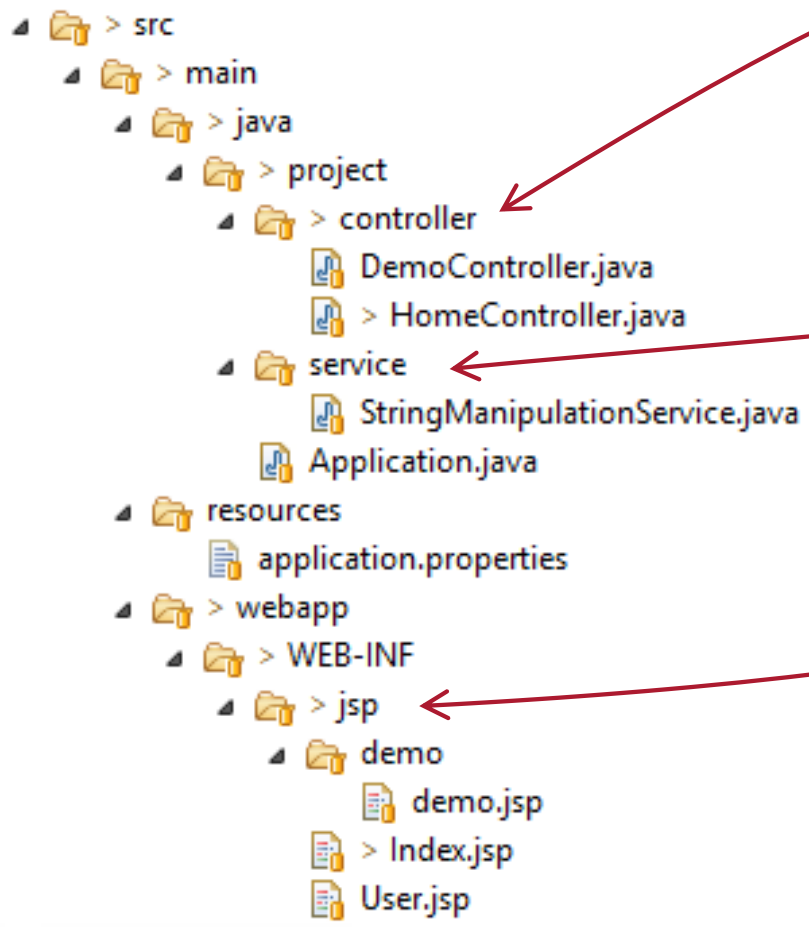
- Web page (HTML) or
- Data (e.g. JSON)

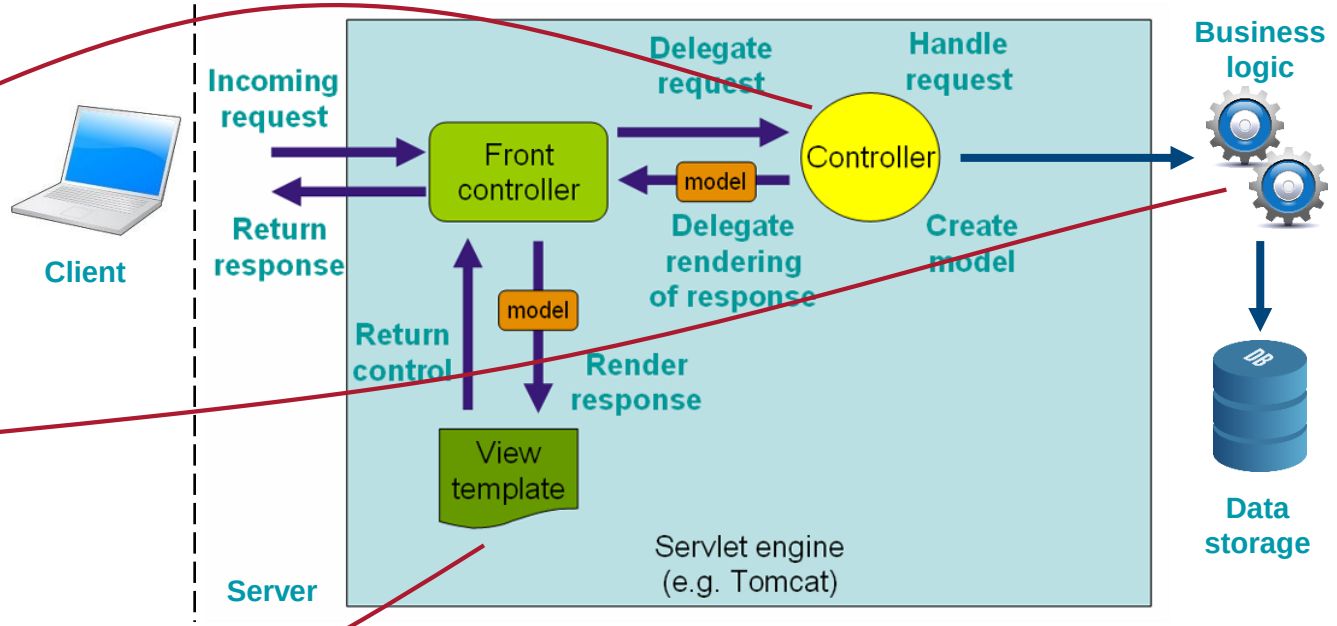
Spring Web MVC Framework



Spring Web MVC Framework

- Typical project structure:





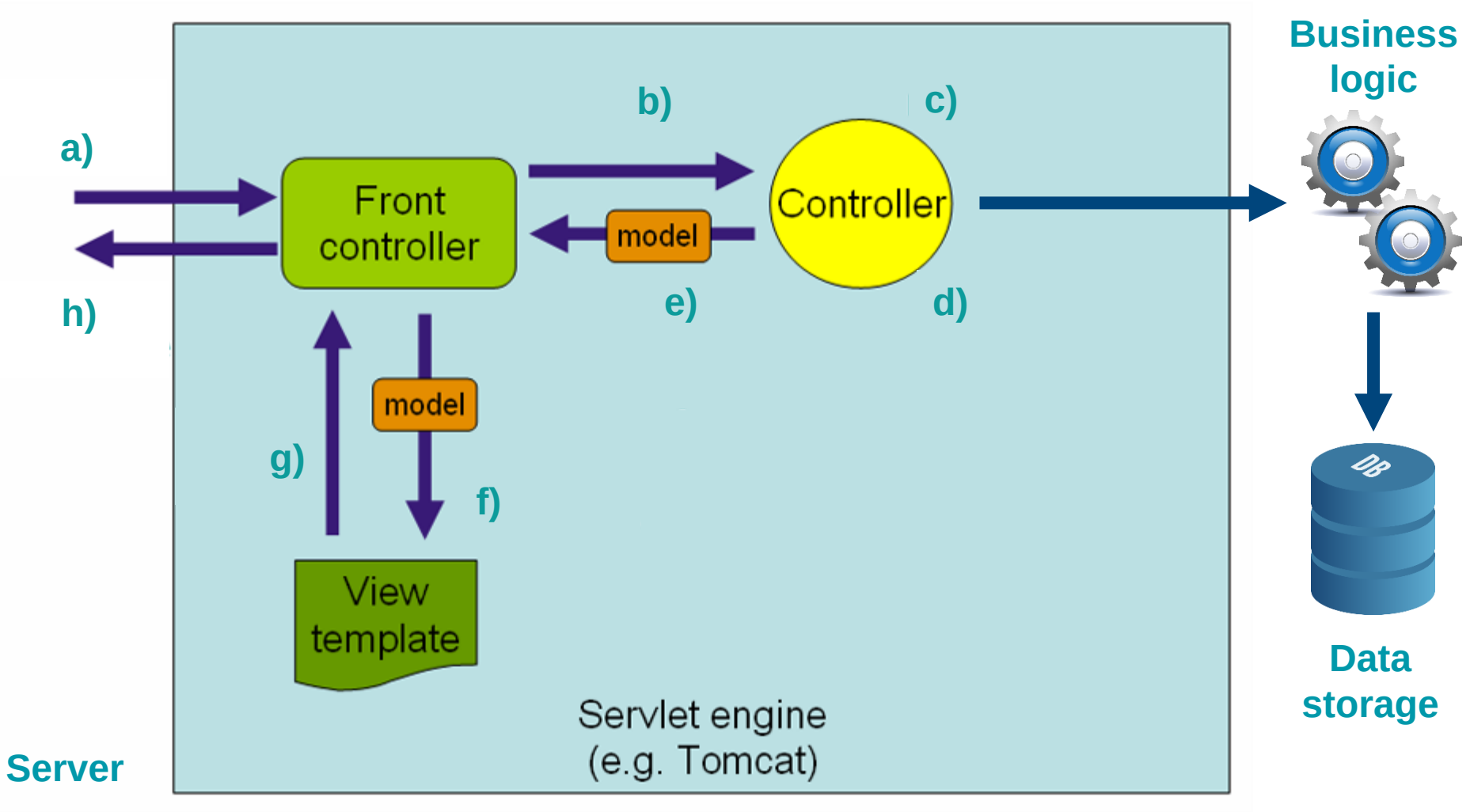
- *See Spring Boot intro slides and skeleton project in Verkefni folder in Ugla for setup details*

In-Class Quiz #4: Spring Web MVC Framework

Which label (1-8) belongs to which arrow (a-h)?

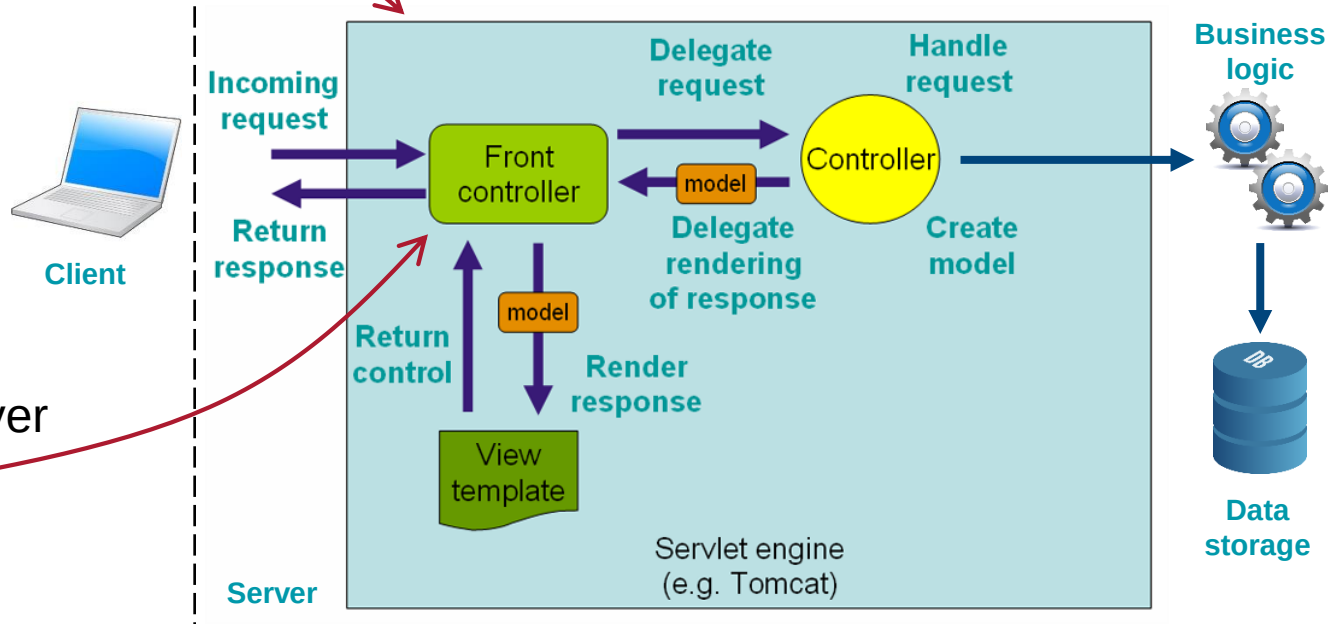
Client

1) Create model
2) Delegate rendering of response
3) Delegate request
4) Handle request
5) Return control
6) Receive request
7) Return response
8) Render response



Setting up the Servlet Engine and Front Controller

- We need a servlet engine that will execute our web application
  - Usually, we'd have to
    - install an application server
    - package our app's files into a WAR
    - deploy WAR on the application server
- We need a Front Controller that will receive requests and distribute them to the appropriate business controller
  - Usually, we'd have to instantiate and configure a Spring `DispatcherServlet`
    - or write one ourselves from scratch
- Spring Web MVC and Spring Boot simplify these steps
  - Java annotations indicate our intentions without having to implement them



The Spring Boot Application Class

```
package project;
// [import statements]

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Implies varies configuration and bootstrapping activities performed by the Spring Boot framework

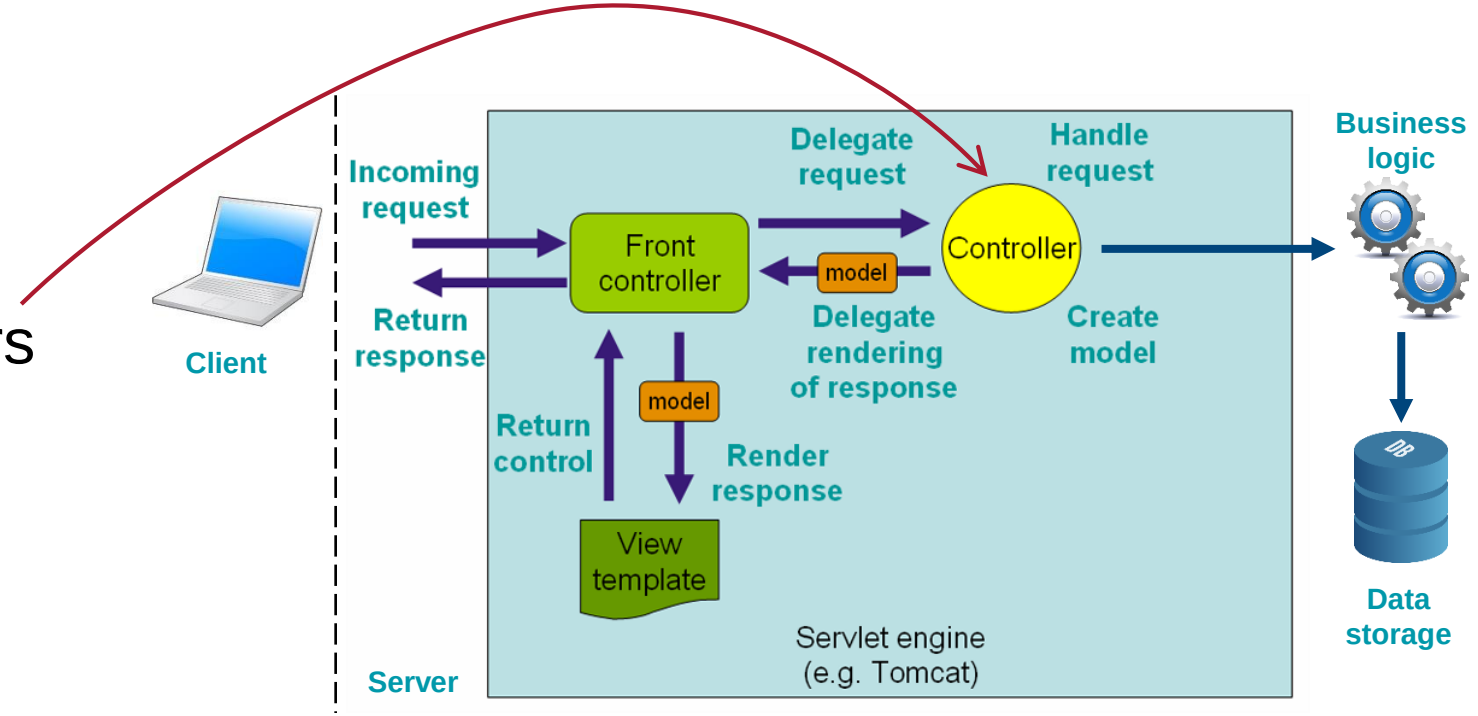
Run the `main` method just like you would run any other Java program, and Spring will

- start an embedded application server
- deploy the web application

Application will then be accessible at `http://localhost:8080` through your browser

Handling Requests

- Clients can trigger different behaviors by requesting different URIs
- We need business controllers that will respond to specific requests
  - Usually, we'd have to
    - Write the controller class
    - Configure the mapping from a particular request URI to a particular controller



- Spring Web MVC simplifies this
  - By providing the `@Controller` and `@RequestMapping` annotations
  - Letting us focus on implementing what the controller does, not how it is wired into the architecture

A Trivial Controller

```
package project.controller;
// [import statements]

@Controller
public class HomeController {

  @RequestMapping(value = "/", method = RequestMethod.GET)
  public String home() {
    // [invocations of application logic]
    return "Index";
  }

}
```

Tells Spring framework to treat this class as an MVC controller

Path that this controller method should be mapped to
- here: `http://localhost:8080/` will invoke this method

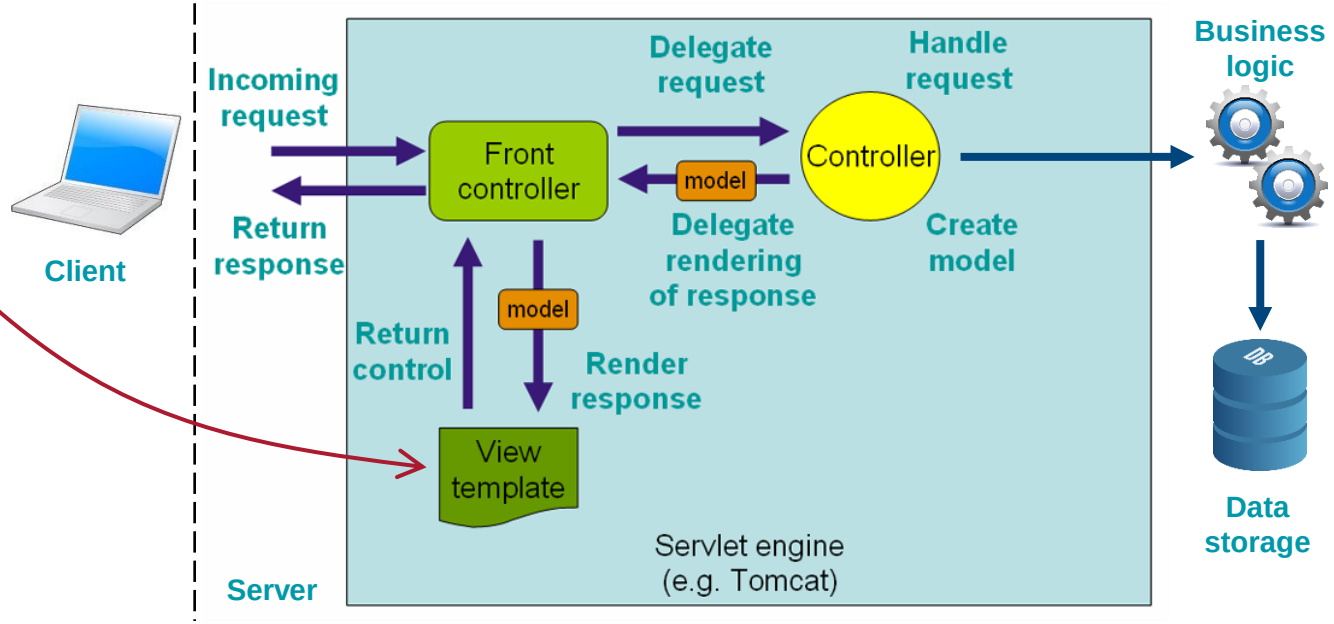
HTTP request type that this controller shall react to
- Links are GET requests
- Form submissions can be GET or POST requests

Path to view template from which response shall be constructed
- here: `Index.jsp`

Application-specific logic typically invoked here:
- Process request parameter
- Prepare model data to use in view

Constructing Responses

- We need view templates that define what our responses should look like
  - Usually, we'd have to
    - Write a JavaServer Page (JSP) for each response page
    - Forward the HTTP Request to it



- Spring Web MVC simplifies this
  - We still need to write the JSP
    - with support from various templating engines
  - In the controller, we can simply name the JSP that should be displayed, without implementing the mechanics of request forwarding

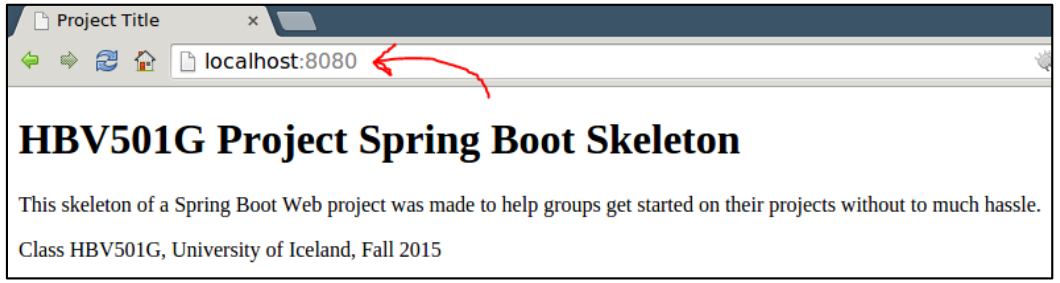
A Static View

Stored on server in `main/webapp/WEB-INF/jsp/Index.jsp`

Configure this location in `resources/application.properties`

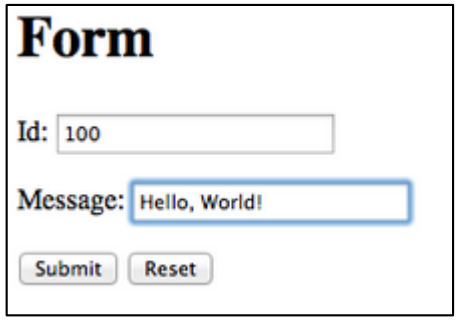
```
<!DOCTYPE html>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html lang="en">
  <head>
      <title>Project Title</title>
  </head>
  <body>
    <h1>HBV501G Project Spring Boot Skeleton</h1>
    <p>This skeleton of a Spring Boot Web project was made to help groups...</p>
  </body>
</html>
```

HTML code of web page to be delivered back to client



Providing Data to the Server

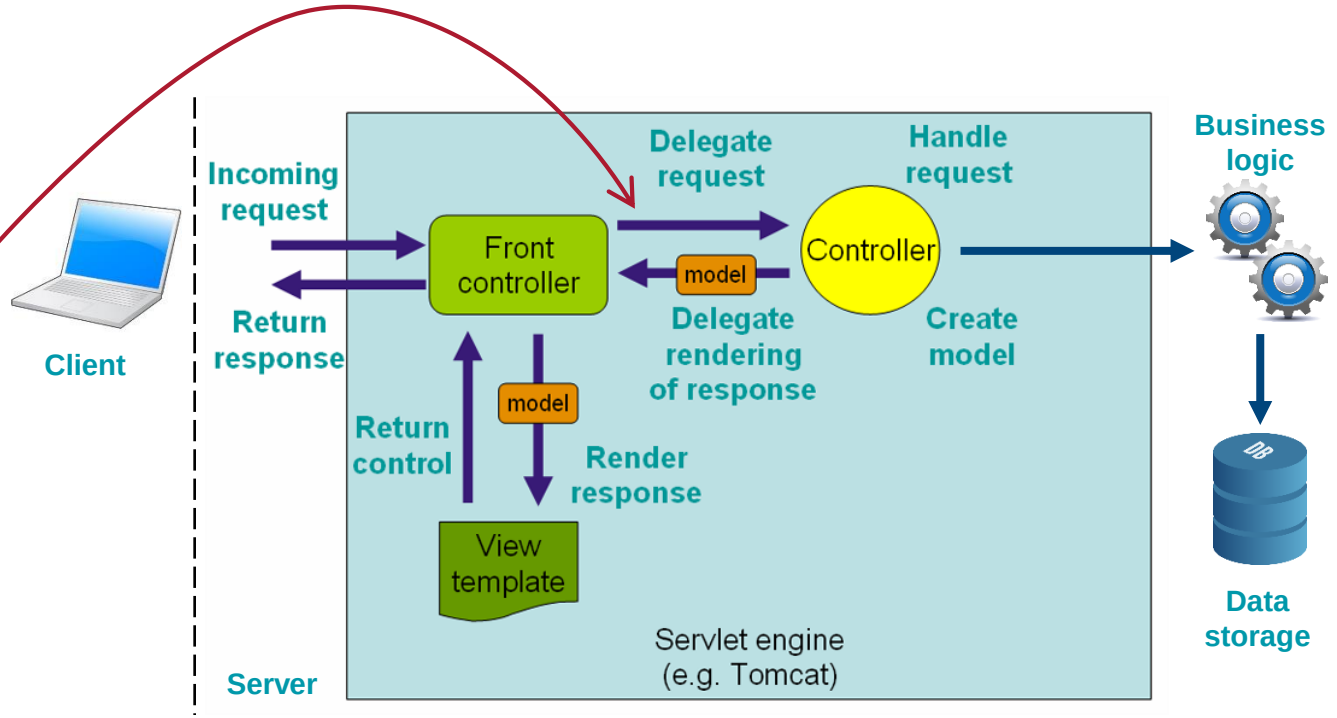
- An HTTP request contains
  - The **address** of the desired resource (URI)
    - e.g. `http://localhost:8080/hello`
    - Used to
      - Identify the resource (file) to be returned to the client (in regular web servers)
      - Identify the endpoint (controller) to handle the request (in application servers)
  - Optional **parameters**
    - Either appended to the URI (in a GET request)
      - e.g. `http://localhost:8080/hello?id=1&message=Hello,+World!`
    - Or submitted in the HTTP request body (in a POST request)
    - Must be extracted and processed by the controller



- Different ways of creating these requests
  - **GET requests:** clicks on links (most common), form submissions, scripts
  - **POST requests:** form submissions (most common), scripts

Receiving Data from the Client

- Clients can add data to requests
  - through web forms or web links
- We want to work with data that clients added to the request
  - Usually, we'd have to
    - extract individual parameters from the URI or HTTP request body



- Spring Web MVC simplifies this
  - In the request handler, we can simply name the request parameters that should be extracted and provided as method parameters
  - Or we can interpret certain segments of the URI as path parameters

Request Parameters

```
package project.controller;
// [import statements]
@Controller
public class HomeController {
@RequestMapping(value="/hello")
  public String hello(
      @RequestParam(value="name", required=false, defaultValue="User") String nafn,
      Model model) {
    // [application logic]
    model.addAttribute("nom", nafn);
    return "Greeting";
  }
}
```

Requesting `http://localhost:8080/hello` will invoke this method

Name of request parameter whose content shall be assigned to method parameter

- here: expected URL format: `http://localhost:8080/hello?name=Jon`
- contents of `name` will be assigned to `nafn`

Value to assign if parameter is not specified in URL

Make data available in data model

- here: contents of local variable `nafn` will be available in data model under the label `nom`

For illustration of namespaces only – typically, you would choose the same label for `name`, `nafn` and `nom`

Path Parameters

```
// [Controller declaration]

@RequestMapping(path="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
public String findPet(
    @PathVariable String ownerId, @PathVariable String petId, Model model) {
  Owner owner = ownerService.findOwner(ownerId);
  Pet pet = owner.getPet(petId);
  model.addAttribute("pet", pet);
  return "DisplayPet";
}
```

Template for extraction of path parameters from mapped URI

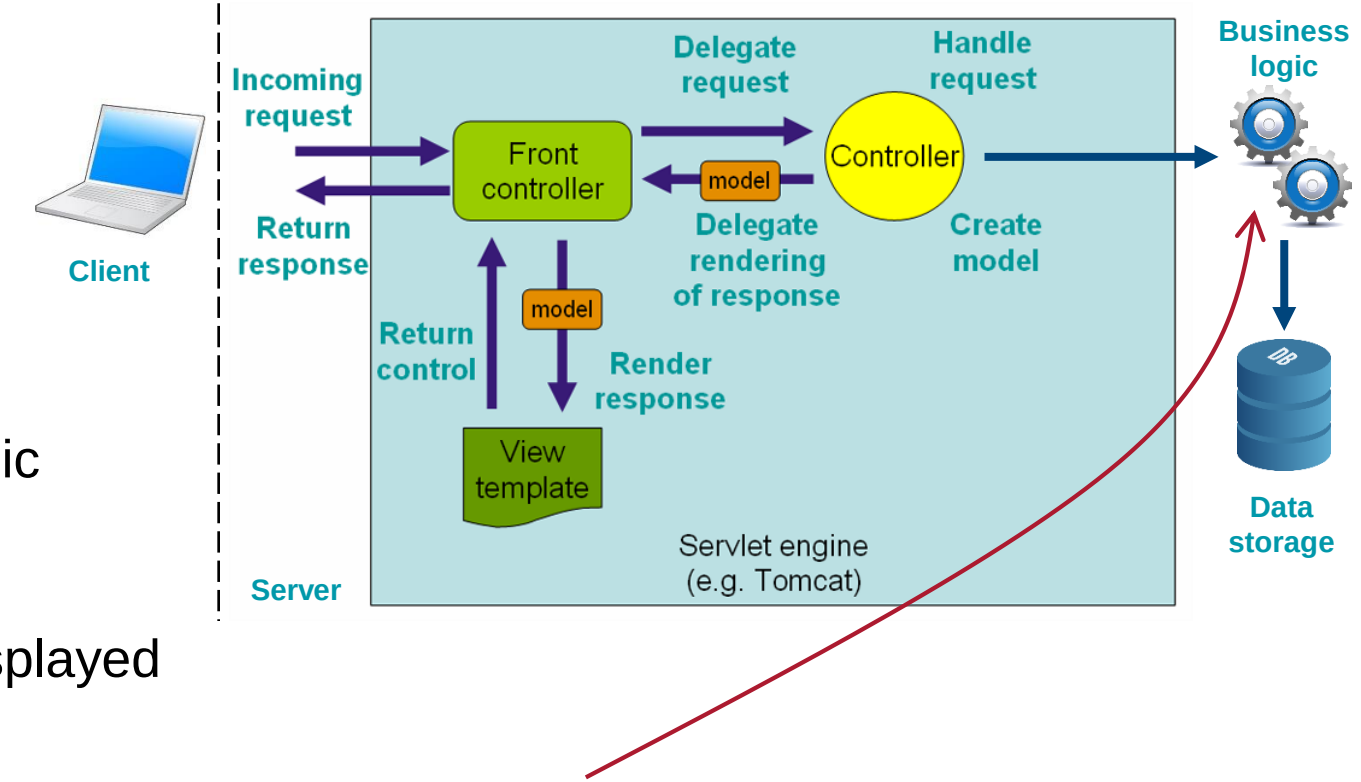
Method parameters with matching names

- Store the pet in the model
- Delegate to view that will display it

Invoking business logic functions

Invoking Business Logic

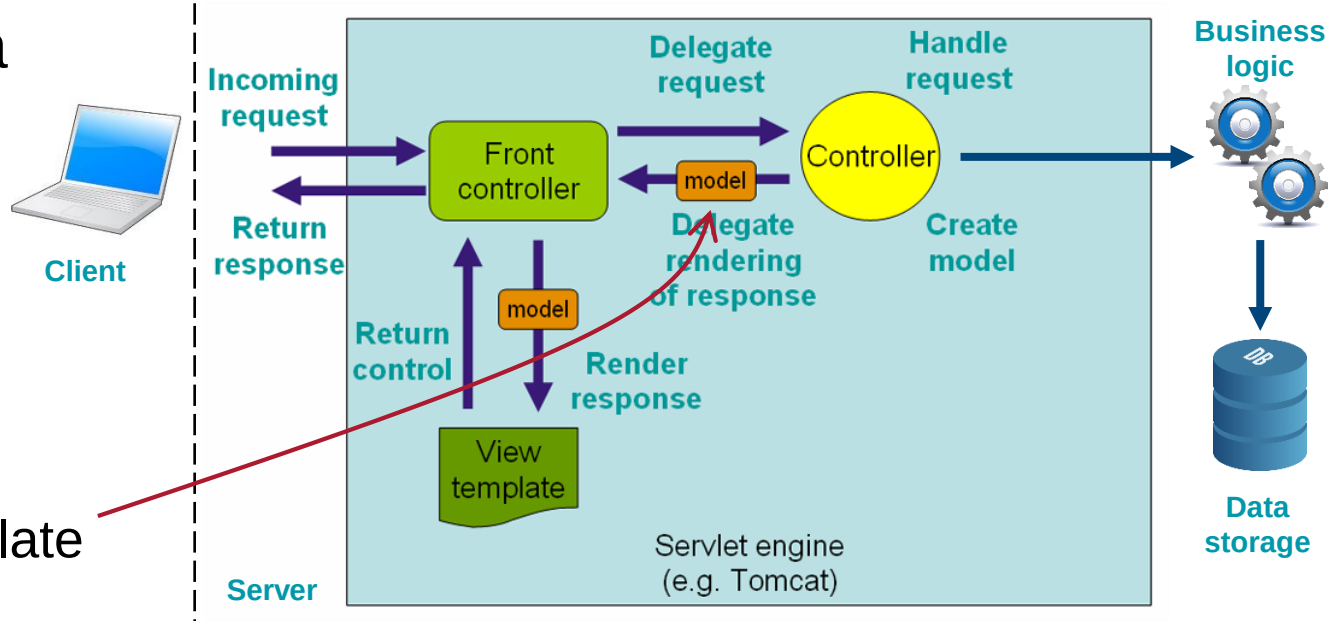
- Usually, the controller should not perform the actual business logic by itself, but deal only with
  - Collecting the necessary inputs from the request, session etc.
  - Calling the intended application logic
  - Preparing the model data required by the view
  - Determining the next view to be displayed



- Application logic can be implemented in plain Java classes and simply invoked from the controller
  - Note: Several invocations in parallel threads are likely! Consider thread-safety issues.

Incorporating Model Data into the View

- We want to include business data into the constructed responses
  - Usually, we'd have to extract data from various sources in the view template
- Spring Web MVC simplifies this
  - In the request handler, we can populate a `Model` with all information that is required by the view to be constructed



Displaying Model Data

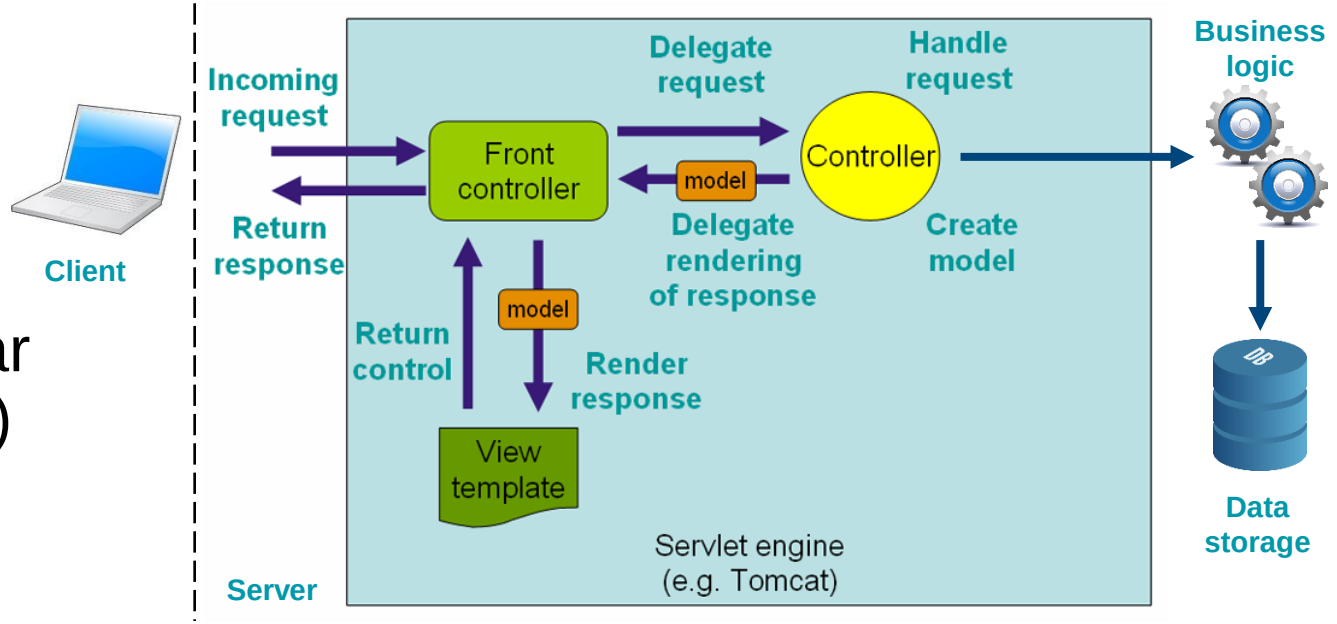
Stored on server in
`main/webapp/WEB-INF/jsp/Greeting.jsp`

```
<!DOCTYPE html>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html lang="en">
  <head>
      <title>Greetings!</title>
  </head>
  <body>
    <p>Hello ${nom}!</p>
  </body>
</html>
```

Integrate model contents into web page
- here: insert contents of model element `nom`

Handling More Complex Form Data

- Usually, we don't want to work on the level of individual request parameters
- Often, users are working with forms that correspond to particular data structures (business objects) on the server



- Spring Web MVC simplifies this
  - by allowing us to refer to JavaBeans implementing the business objects directly
    - in the formulation of the form
    - in the handling of the request

JavaBean for Storing Form Data

```
package hello;

public class UserInfo {
    private long id;
    private String email;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

Simple JavaBean (i.e. a plain old Java object with private attributes and public getters and setters) describing the data we want to capture in the form

- here: information about a user, e.g. an ID and e-mail address

Controller Distinguishing Requests for Form and Result

```
package hello;
// [import statements]
@Controller
public class UserInfoController {
    @RequestMapping(value="/info", method=RequestMethod.GET)
    public String userInfoForm(Model model) {
        model.addAttribute("userinfo", new UserInfo());
        return "InfoForm";
    }
    @RequestMapping(value="/info", method=RequestMethod.POST)
    public String userInfoSubmit(@ModelAttribute UserInfo userInfo, Model model) {
        model.addAttribute("userinfo", userInfo);
        return "InfoResult";
    }
}
```

Under the URI `/info`, two different behaviors and resulting views shall be accessible:

- Upon a GET request (i.e. a regular link to the page), show the page `InfoForm.jsp`
- Upon a POST request (i.e. a submission of the form), show the page `InfoResult.jsp`

- Creates a new `UserInfo` object
- Adds it to the model
- Delegates to `InfoForm.jsp`

- Retrieves `UserInfo` object from request (with attributes populated through the form)
- Adds it to the model
- Delegates to `InfoResult.jsp`

Form View Template

Stored on server as `InfoForm.jsp`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Handling Form Submission</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <h1>Form</h1>
    <form action="#" th:action="@{/info}" th:object="${userinfo}" method="post">
      <p>Id: <input type="text" th:field="*{id}" /></p>
      <p>e-Mail: <input type="text" th:field="*{email}" /></p>
      <p><input type="submit" value="Submit" /> <input type="reset" value="Reset" /></p>
    </form>
  </body>
</html>
```

Submit form data to controller at URL `/info`

Populate form with data from object stored under `userinfo` in model

Submit form data in HTTP POST request

Populate form fields with attributes `id` and `email` of `userinfo` object

Result View Template

Stored on server as `InfoResult.jsp`

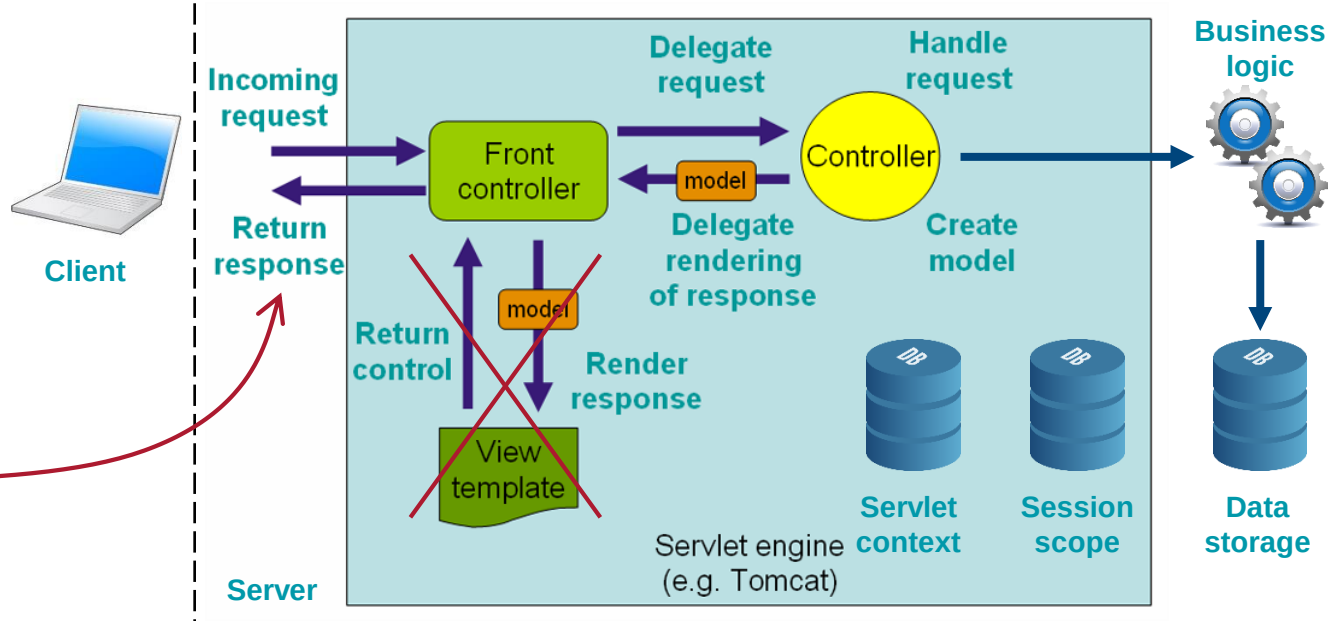
```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Handling Form Submission</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <h1>Result</h1>
    <p th:text="'Id: ' + ${userinfo.id}" />
    <p th:text="'e-Mail: ' + ${userinfo.content}" />
    <a href="/info">Return to form</a>
  </body>
</html>
```

Get attributes `id` and `content` from object found under label `userinfo` in model, and integrate into HTML code

Trigger HTTP GET request

Returning Data Instead of a View

- When implementing a REST API, we don't need a view template to construct a web page
- Instead, the controller should create JSON or XML data that is returned directly in the HTTP Response



- Spring Boot simplifies this
  - by enabling us to bypass the view template
  - and automatically converting JavaBeans to a JSON representation

JavaBean Containing Data to Transfer

```
package hello;

public class Greeting {

    private final long id;
    private final String content;

    public Greeting(long id, String content) {
        this.id = id;
        this.content = content;
    }

    public long getId() { return id; }
    public String getContent() { return content; }
}
```

Plain old Java object with constructor and accessor methods describing the data we want to transmit

REST Controller

```
package hello;
// [import statements]

@RestController
public class GreetingController {

  private static final String template = "Hello, %s!";
  private final AtomicLong counter = new AtomicLong();

  @RequestMapping("/greeting")
  public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {
    return new Greeting(counter.incrementAndGet(),
                        String.format(template, name));
  }
}
```

Indicates that this controller's request handlers will not determine a view template, but return data for inclusion in the HTTP response directly

Example request:
`http://localhost:8080/greeting?name=Jon`

Resulting HTTP response body:

```
{
  "id": 1,
  "content": "Hello, Jon!"
}
```

Request processing (URI mapping, parameter extraction, session access etc.) works as usual

- Construct and return the object to be sent to the client.
- Spring will use the Jackson library to automatically convert this object to JSON and include it in the response body.