Desarrollo de aplicaciones Web con Spring MVC

Jesús Arias Fisteus

Computación Web (2024/25)

uc3m | Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática

Parte I

El patrón Modelo Vista Controlador

El patrón Modelo–Vista–Controlador

El patrón **Modelo–Vista–Controlador (MVC)** es un patrón de diseño de *software* para interfaces de usuario que divide la aplicación en tres componentes: **modelo**, **vista** y **controlador**.

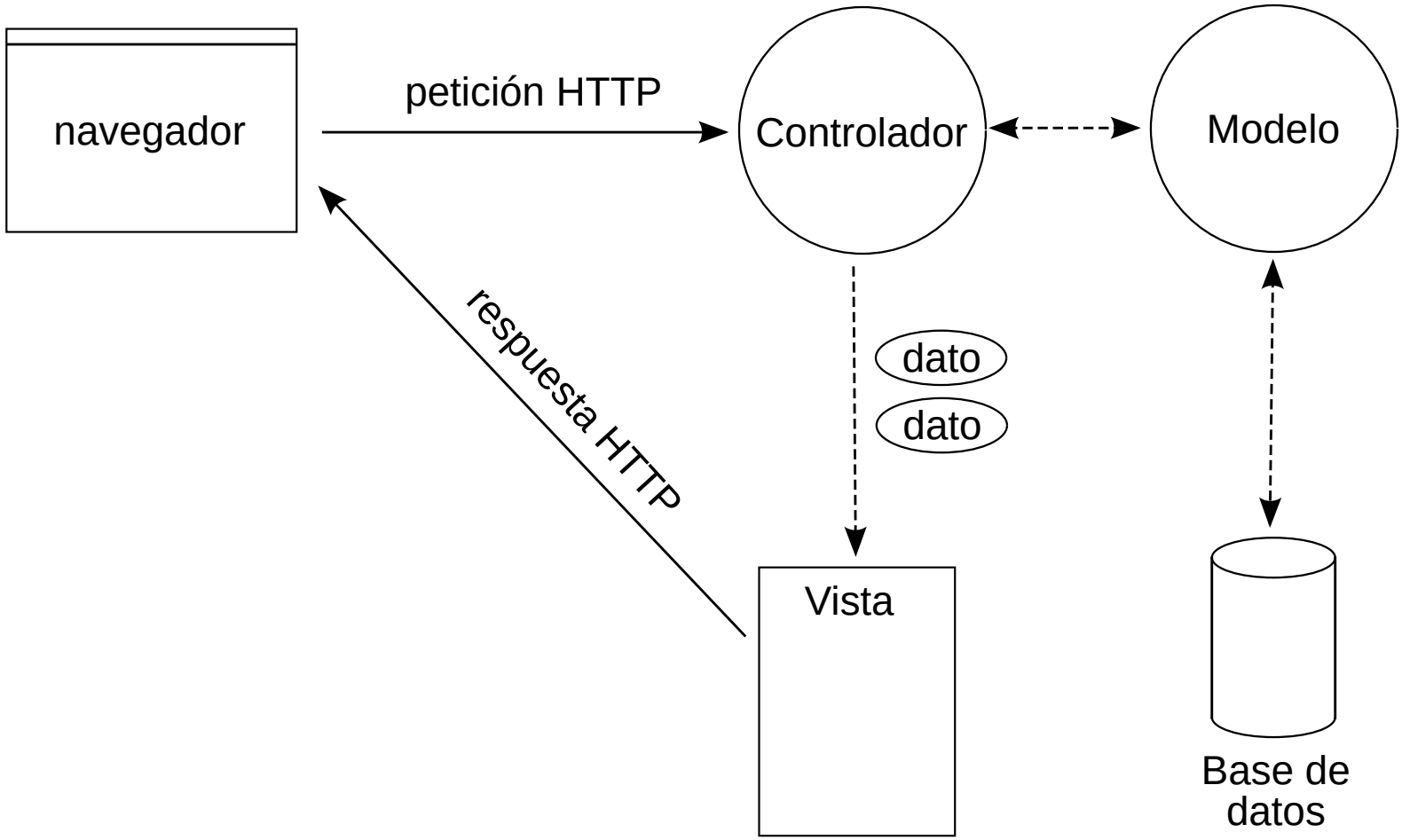
El patrón Modelo–Vista–Controlador

El componente del **modelo** gestiona los datos, lógica y reglas de la aplicación, incluyendo normalmente, en el caso de las aplicaciones Web, el almacenamiento de datos en una base de datos.

El componente de la **vista** presenta la información a los usuarios (normalmente con HTML y CSS en el caso de las aplicaciones Web) y recoge la entrada de los usuarios.

El componente del **controlador** acepta la entrada de los usuarios e invoca la funcionalidad apropiada en los componentes del modelo y de la vista.

El patrón MVC en aplicaciones Web



El patrón MVC en aplicaciones Web

1. Mediante una página HTML creada por la vista anterior, el cliente envía una petición HTTP al servidor (pinchando en un hipervínculo, enviando un formulario, etc.).
2. El controlador recibe la petición e interacciona con el modelo.
3. El controlador invoca a la vista con los datos que necesita presentar al usuario.
4. La vista recoge dichos datos, crea la página HTML que los presenta y construye la respuesta HTTP a enviar al cliente.

Parte II

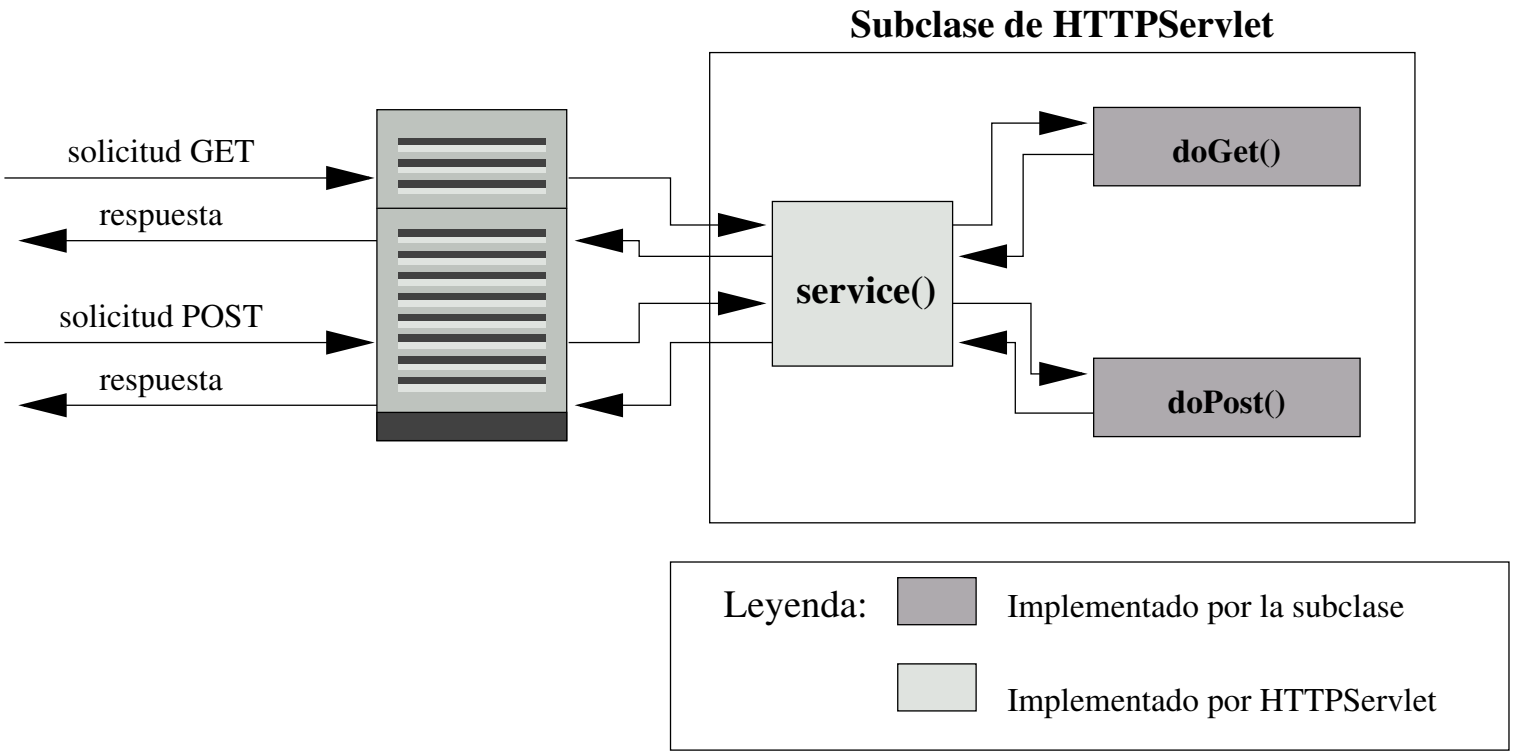
Servlet

Servlet

Un **Servlet** es un programa Java que se ejecuta en un servidor, normalmente de HTTP, que atiende peticiones de los clientes y genera respuestas.

Un **contenedor de servlets** gestiona su ejecución y le proporciona acceso a la API de Servlet.

Servlets de HTTP



Ejemplo de servlet

```
public class HolaMundo extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
            throws IOException, ServletException {
      String name = request.getParameter("name");
      String safeName = cleanXSS(name);
      response.setContentType("text/html; charset=UTF-8");
      PrintWriter out = response.getWriter();
      out.println("<!DOCTYPE html>");
      out.println("<html>");
      out.println("<head>");
      out.println("<title>¡Hola!</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<h1>¡Hola " + safeName + "!</h1>");
      out.println("</body>");
      out.println("</html>");
  }
}
```

Sesiones

Las **sesiones** permiten a los servlets mantener información sobre un usuario a lo largo de varias peticiones HTTP.

Cada sesión se identifica mediante un **token de sesión**, un identificador aleatorio, criptográficamente seguro y único que se envía al cliente, y el cliente envía de vuelta en cada nueva petición. Normalmente se almacena en una **cookie**.

Cada sesión se asocia a un objeto **HttpSession** mediante el cual la aplicación puede guardar y recuperar objetos Java, como por ejemplo la identidad del usuario autenticado.

Filtros

Los **filtros** son programas que complementan a los servlets con la capacidad de interceptar y modificar peticiones antes de que lleguen a un servlet o respuestas después de que salgan de él.

(Por ejemplo, se puede programar un filtro que intercepte las peticiones a uno o más servlets para comprobar previamente que el usuario autenticado esté autorizado a acceder al mismo.)

Parte III

Spring MVC

Spring MVC

Spring MVC[1] es un *framework* que permite el desarrollo de aplicaciones Web que siguen el patrón MVC sobre **Spring Framework**.

Spring Framework[2] es un entorno para el desarrollo de aplicaciones empresariales para diversos entornos (no solo Web) sobre la plataforma de Java, en lenguajes como Java, Kotlin o Groovy.

[1] `https://docs.spring.io/spring-framework/reference/web/webmvc.html`

[2] `https://spring.io/projects/spring-framework`

Spring Boot

Spring Boot[3] es una extensión de Spring Framework que proporciona un entorno de ejecución de aplicaciones para Spring Framework preconfigurado conforme a distintos convenios y buenas prácticas, con el objeto de simplificar el desarrollo y despliegue de las aplicaciones.

[3] `https://spring.io/projects/spring-boot`

El componente del modelo

El componente del **modelo** suele construirse en *frameworks* Web sobre bibliotecas de persistencia de objetos, conocidas normalmente como ORM[4], las cuales proporcionan automáticamente el código necesario para **crear**, **leer**, **actualizar** y **borrar** (CRUD) objetos en la base de datos.

[4] ORM: *mapeo objeto relacional*.

El componente del modelo

Sin embargo, también existen alternativas a los ORM ampliamente usadas, como son:

- Construir consultas SQL directamente desde el código de la aplicación (por ejemplo, en Java, sobre JDBC).
- Usar bases de datos no basadas en el modelo relacional ni en el lenguaje SQL (por ejemplo, MongoDB).

El componente del modelo en Spring MVC

El componente del modelo en Spring MVC se suele construir sobre **Spring Data JPA**[5], el cual trabaja, a su vez, sobre implementaciones de **Jakarta Persistence**[6] (anteriormente conocido como Java Persistence API o JPA).

La implementación por defecto de Jakarta Persistence en Spring es **Hibernate ORM**[7].

[5] https://spring.io/projects/spring-data-jpa
[6] https://jakarta.ee/specifications/persistence/
[7] https://hibernate.org/orm/

El componente del modelo con Spring Data JPA

```
/*
 * Declaración de una clase del modelo
 */
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Column(nullable = false)
    @NotBlank
    @Size(max = 64)
    private String name;

    @Column(unique = true, nullable = false)
    @Email
    @NotBlank
    @Size(max = 128)
    private String email;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }
    (...)
}
```

El componente del modelo con Spring Data JPA

```java
/*
 * Declaración de una clase repositorio
 */
public interface UserRepository
        extends CrudRepository<User, Integer> {
    User findByEmail(String email);
}

// Inserción de un objeto
User user = new User();
user.setName("Mary");
user.setEmail("mary@example.com");
userRepository.save(user);

// Obtención de un objeto desde la base de datos
User user = userRepository.findByEmail("mary@example.com");
```

El componente de la vista

Las **vistas** suelen ser construidas en *frameworks* Web mediante **plantillas** HTML que, a partir de un conjunto de variables que contienen los datos a mostrar, producen la página HTML que los presenta.

Las plantillas consisten fundamentalmente en marcado HTML con pequeños fragmentos de código incrustado para inyectar datos, mostrar contenido condicionalmente, iterar sobre los datos de una colección, etc.

El componente de la vista en Spring MVC

Las aplicaciones Spring MVC usan normalmente los lenguajes de plantillas **Thymeleaf**[8] o **Java Server Pages (JSP)**.

[8] `https://www.thymeleaf.org/`

Thymeleaf

```
<div class="messages">
    <div class="message" th:each="post␣:␣${posts}">
        <div class="text" th:text="${post.getText()}">
            Text of the post
        </div>
        <div class="metadata">
            <span class="author"
                th:text="${post.getUser().getName()}">
                    Author's name
            </span>
            <span class="date"
                th:text="${post.getTimestamp()}">
                    Timestamp
            </span>
        </div>
    </div>
</div>
```

El componente del controlador

Normalmente se programan los **controladores** mediante estructuras normales de código (métodos / funciones) en el lenguaje de programación del *framework*.

El componente del controlador

Cada método o función se mapea a una o más rutas de peticiones HTTP, recibe todos los datos que necesite de la petición HTTP, interacciona con el modelo y, finalmente, selecciona la plantilla a ser presentada y le pasa a esta los parámetros que necesite.

Cuando el *framework* recibe una petición HTTP, decide, conforme a la ruta de la petición, qué método o función del controlador necesita ser invocado.

Controladores en Spring MVC

```
@Controller
@RequestMapping(path = "/")
public class MessageController {

  @Autowired
  private MessageRepository messageRepository;

  @GetMapping(path = "/message/{messageId}")
  public String viewMessage(
      @PathVariable("messageId") int messageId,
      Model model) {
    Message message = messageRepository.findById(messageId);
    if (message != null) {
      model.addAttribute("message", message);
      return "view_message";
    } else {
      throw new ResponseStatusException(
        HttpStatus.NOT_FOUND, "Message not found");
    }
  }
  (...)
}
```

Controladores en Spring MVC

```
@PostMapping(path = "/post")
public String postMessage(
        @ModelAttribute Message message,
        Principal principal) {
    User user =
        userRepository.findByEmail(principal.getName());
    message.setUser(user);
    message.setTimestamp(new Date());
    messageRepository.save(message);
    return "redirect:message/" + message.getId();
}
```

Controladores en Spring MVC

Los **controladores** de Spring MVC **se ejecutan como servlets** en las capas subyacentes del *framework*.

Parte IV

Leyendo datos desde formularios

Controles en formularios HTML

Cuando el usuario presiona el botón de envío de un formulario HTML, el navegador lee los valores de todos los controles del mismo y los encapsula en una petición HTTP como una **colección de parámetros nombre–valor**. La URL de la petición se toma del atributo **`action`** del elemento **`form`**.

El **nombre** de cada parámetro se toma del atributo **`name`** del control correspondiente.

El **valor** de cada parámetro se toma de la entrada del usuario en dicho control.

Controles en formularios HTML

```
<form th:action="@{/find-user}" method="get">
  <label>
    Search user:
    <input type="email" name="email"
        placeholder="email␣address">
  </label>
  <input type="submit" value="Search">
</form>
```

Lectura de parámetros de la petición en Spring MVC

Spring proporciona a los métodos del controlador los parámetros de la petición HTTP directamente como argumentos del método si estos se declaran mediante la anotación **@RequestParam**.

```
@GetMapping(path = "/find-user")
public String findUser(
        @RequestParam(name = "email") String email,
        Model model) {
    User user = userRepository.findByEmail(email);
    model.addAttribute("user", user);
    return "view_user";
}
```

Lectura de parámetros de la petición en Spring MVC

También es posible recibir como argumento un **bean**[9], cuyos atributos se inicializan con los valores de los parámetros de la petición que coincidan en nombre con los mismos, mediante la anotación **@ModelAttribute**.

```
@PostMapping(path = "/signup")
public String signUp(@ModelAttribute("user") User user) {
  if (userRepository.findByEmail(user.getEmail()) != null) {
    return "redirect:register?duplicate_email";
  }
  userService.register(user);
  return "redirect:login?registered";
}
```

[9] Los *beans* son clases Java que encapsulan datos (propiedades) y siguen el convenio de proporcionar un constructor sin argumentos y métodos de acceso (*getters* y *setters*) a las propiedades.

Parte V

Plantillas Thymeleaf

Expresiones

Se inyecta el resultado de evaluar una expresión en un lugar concreto de la página HTML mediante la sintaxis ${}.

```
<p>
    Your name is <span th:text="${user.name}">John</span>
    and you are from
    <span th:text="${user.country}">Nowhere</span>.
</p>
```

(Los valores "John" y "Nowhere" se mostrarán solo si se abre la plantilla directamente en un navegador, sin ejecutarla. Si se ejecuta la plantilla pidiendo el recurso al servidor, son reemplazados por el resultado de cada expresión.)

Bucles

Se definen bucles sobre colecciones de datos con la sintaxis **th:each**. El elemento que contenga dicha directiva se presenta tantas veces como elementos tenga la colección.

```
<ul>
    <li th:each="user : ${users}" th:text="${user.name}">
        John Doe
    </li>
</ul>
```

```
<div class="message" th:each="message : ${messages}">
    <div class="text" th:text="${message.getText()}">
        Text of the post
    </div>
    <div class="metadata">
        <span class="author"
              th:text="${message.getUser().getName()}">
            Author's name
        </span>
    </div>
</div>
```

Condicionales

Se definen condicionales con la sintaxis **th:if**. El elemento que contenga dicha directiva solo aparece si la condición se evalúa a valor cierto.

```
<p>
    <span th:text="${user.getName()}">John Doe</span>
    <span th:if="${user.isAdmin()}">
        (admin)
    </span>
</p>
```

Parte VI

Rutas relativas

Rutas relativas

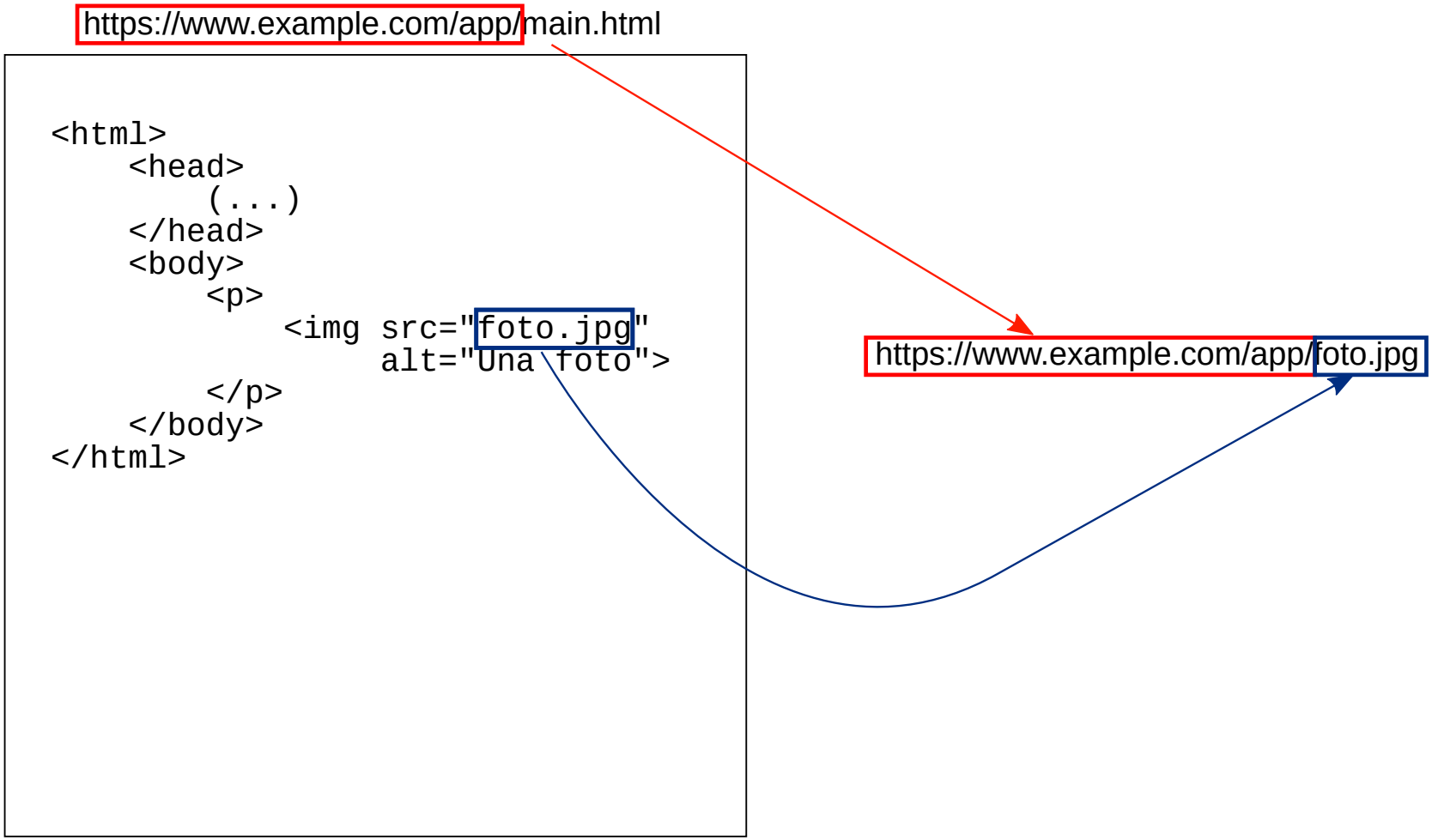
Una aplicación web generalmente consiste en muchos recursos, tanto estáticos como dinámicos. Sus URLs generalmente comparten el mismo esquema y autoridad, es decir, suele accederse a ellas a través del mismo protocolo, nombre de dominio y puerto TCP.

Una página HTML generalmente hace referencia a otros recursos en la misma aplicación web a través de hipervínculos, formularios, imágenes, referencias a hojas de estilo, referencias a archivos JavaScript, etc.

Rutas relativas

En una página HTML a la cual se accede a través de una URL dada, las **rutas relativas** permiten a los creadores evitar tener que especificar URLs completas, proporcionando simplemente la parte que cambia con respecto a la URL de la página HTML actual.

Rutas relativas



Rutas relativas

En la mayoría de los casos se debería usar rutas relativas para hacer referencia a otras URLs de la misma aplicación:

- Las páginas HTML son más compactas y fáciles de leer, y los errores en URLs son menos frecuentes.
- Se pueden mover las aplicaciones a otro esquema o autoridad sin necesidad de actualizar los enlaces en cada página HTML.
- Durante la fase de desarrollo, se puede acceder a los recursos en despliegues de la aplicación en local o en servidores que no son de producción.

Referencias

- Shameer Kunjumohamed, Hamidreza Sattari, Alex Bretet, Geoffroy Warin. *Spring MVC: Designing Real-World Web Applications*, Packt Publishing (2016):
  - Online access at O'Reilly through UC3M Library