Day 13

Sending Data to Web Application

- Data can be sent from the client to the web application
- Use HTTP POST method to send data to the web application
  - Data/payload is transport in the body of the request
    - After the last HTTP header
  - HTTP header `Content-Type` specifies the payload format/encoding
- POST is used to create/insert data
  - Eg a new order entry, where the payload is the detail of the order

Form

`application/x-www-form-urlencoded`

Structured data

`application/json, application/xml`

Large binary file

`multipart/form-data`

HTTP POST Request

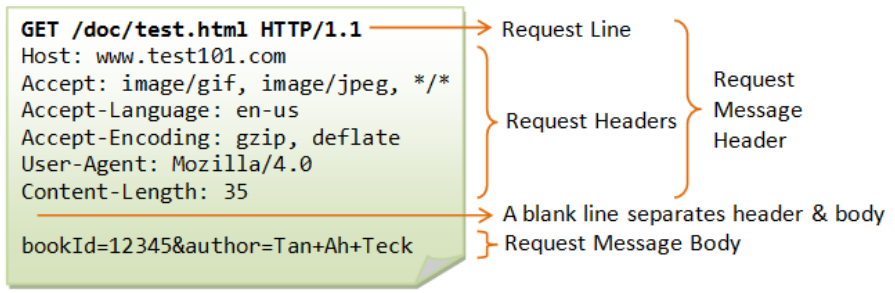
```
<form method="POST" action="/user">
   <input type="text" name="name">
   <input type="email" name="email">
   <input type="tel" name="phone">
   <button type="submit">
</form>
```

Form field are encoded in `x-www-form-urlencoded`

```
POST /user
Accept: text/html
Content-Type: application/x-www-form-urlencoded

name=fred&email=fred@gmail.com&phone=12345678
```

HTTP POST



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

Post method

Payload encoding

```
POST /user
Accept: text/html
Content-Type: application/x-www-form-urlencoded
```

```
name=fred&email=fred@gmail.com&phone=12345678
```

Specifies the result encoding that the client is expecting

Payload

Mapping Form Fields - `MultiValueMap`

```
name=fred&email=fred@gmail.com&phone=12345678
```

Use `@RequestBody` to map payload to `MultiValueMap`

All form fields are added to the Map
More appropriate if there are lots of inputs from the from

```
@PostMapping(...)
public String createUser(
   @RequestBody MultiValueMap<String, String> form,
   Model model) {

      String name = form.getFirst("name");
      String email = form.getFirst("email");
      String phone = form.getFirst("phone");
      ...
}
```

Mapping Form Fields - @ModelAttribute

```
POST /user
Accept: text/html
Content-Type: application/x-www-form-urlencoded

name=fred&email=fred@gmail.com&phone=12345678

@Controller
@RequestMapping(path="/user")
public class UserController {
   @PostMapping(
      consumes = "application/x-www-form-urlencoded",
      produces = "text/html")
   public String createUser(@ModelAttribute User user, Model model) {
      // process the data
      ...
   }
}
```

User object is created from form fields

SpringBoot instantiates `User`, injects the form fields into the object and passes it to the request handler

Validating Form Inputs

- Many forms require inputs to adhere to a set of constraints
  - E.g. Names must be longer than 5 characters, emails must be in a proper format, cannot proceed if age is less than 18
- Spring Boot provides a set of annotations for specifying these constraints, for annotating the models
- Validate form inputs by annotating the model with constraints from the `jakarta.validation.constraints` package
- Add the following dependency to `pom.xml`

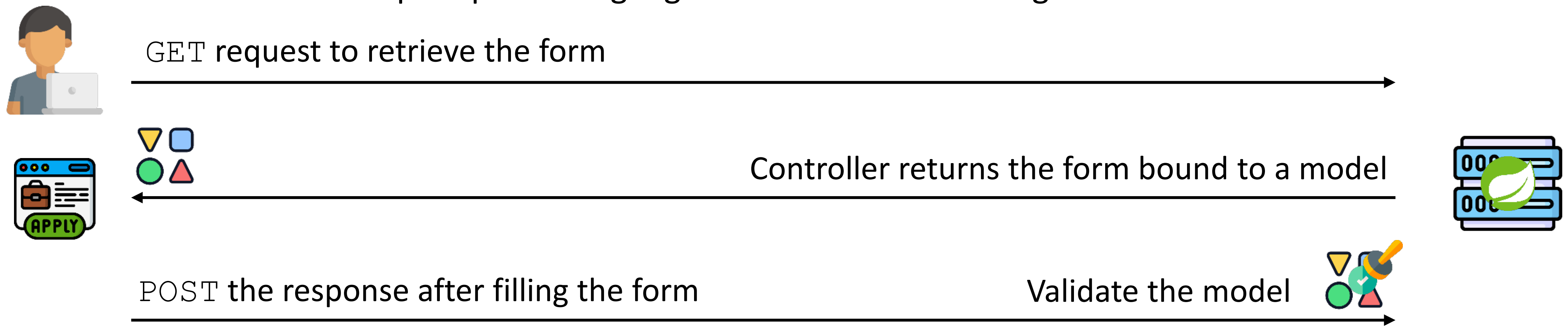
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
    <version> check the version to use </version>
</dependency>
```

Constraints

- Common constraints
  - `@NotNull` – a property must not be null
  - `@NotEmpty` – a property must not be empty
  - `@AssertTrue`, @AssertFalse – property must be true/false
  - `@Min` , `@Max` – numerical properties must be greater than a minimum and less than the specified maximum. `@Min` and `@Max` can be used independently
  - `@Email` – a property must have a properly formatted email address
  - `@Pattern` – the value of a property must match the specified pattern
  - `@Past`, `@PastOrPresent`, `@FutureOrPresent`, `@Future` – Dates in the past, present or future
    - Typically used with `@DateTimeFormat`
  - `@Size` – length of a property eg. String, collections
- Optionally can set a message when the constraint is violated

Validating Form Inputs

- Create a model, annotate properties with constraints
- Handle a `GET` method for the form, and bind the model to the form with `data-th-object`
  - Bind model properties to the form fields `*{}`
- After the form is `POST` back, use the `@Valid` annotation to validate the model
  - If there are any errors, redisplay the form with the errors
- Two types of validation errors
  - Syntactic – e.g. password length, email correctly formatted, etc
  - Semantic – those require processing e.g. if an email has been registered


GET request to retrieve the form
Controller returns the form bound to a model
APPLY
POST the response after filling the form
Validate the model

Example – Annotating Model with Constraints

```
public class Person {
  @NotNull(message="Name cannot be null")
  @Size(min=2, max=32, message="Name must be between 2 and 32 characters")
  private String name;

  @Email(message="Must be a valid email")
  private String email;

  private Boolean married;

  @Min(value=1, message="Age cannot be less than 1")
  @Max(value=100, message="Age cannot be more than 100")
  private Integer age;

  // getters and setters
}
```

Error message

The threshold of the constraint

Constraints on properties

Example – Annotating Model with Constraints

For collections

```
@NotEmpty(message="Life would be more interesting if you have a hobby")
@Size(min=2, message="You should have more than 1 hobby")
private List<String> hobbies;

@Past(message="Are you John Connor?")
@NotNull(message="You must set your DOB")
@DateTimeFormat(pattern="yyyy-MM-dd")
private LocalDate dateOfBirth;
```

Set the date constraint for the `dateOfBirth` property

Example – Binding the Form to a Model

```
@Controller
@RequestMapping(path="/register")
Public class RegisterController {

  @GetMapping
  public String getRegistration(Model model) {
    model.addAttribute("registration", new Registration());
    return "registration";
  }
}
```

Create the model and bind it to the form

Return the form with model bindings

Example – Binding the Form to a Model

```
<form method="POST" action="/register" data-th-object="${registration}">

  <input type="text" data-th-field="*{name}">
  <input type="number" data-th-field="*{age}">


  <input type="checkbox" value="swim" data-th-field="*{hobbies}">
  <input type="checkbox" value="jog" data-th-field="*{hobbies}">
  <input type="checkbox" value="read" data-th-field="*{hobbies}">

  <input type="radio" data-th-field="*{married}">

  <input type="date" data-th-field="*{dateOfBirth}">

  <button type="submit">Register</button>


</form>
```

Bind the model to the form. A form can only be bound to 1 model

Bind the form field to the property of the model with the `*{}`

Set the selected values to the collection

Boolean property, works with either a single checkbox or radio button

Binding a date. Form field can either be `date` or `datetime-local`. The property of date should be `LocalDate` and `LocalDateTime` for the latter

Example – Validating Form Fields

`BindingResult` contains the validation results

Syntactic validation
Validate the data capture from the form by the model

```
@PostMapping
public String postRegistration(@Valid Registration registration
      , BindingResult binding) {

   if (binding.hasErrors())
     return "registration";

   // Check for other errors
   if (!isNameUnique(registration.getName())) {
     FieldError err = new FieldError("registration", "name"
          , "%s is not available".formatted(registration.getName());
     binding.addError(err);
     return "registration";
   }

   return "thankyou";
}
```

If there are validation errors, return to the form and report the errors

Semantic validation
The errors consist of a object name, related field and the error message.
Eg add a `name` field error to `registration`. object
Add the `FieldError` to the `BindingResult` object.
Can add multiple errors with the same name.

Example – Displaying Errors

```
.error {
  color: red;
}
```

```
<form method="POST" data-th-action="@{/register}" data-th-object="${registration}">

    <input type="text" data-th-field="*{name}">
    <p data-th-if="${#fields.hasErrors('name')}"
       data-th-errors="*{name}"
       data-th-errorclass="error"/>

       ...

</form>
```

Check if the property has error; display the error if any. Display the `message` attribute in the constraint annotation with `data-th-errors`

Use the CSS if the property has error

Display all `name` field related errors

`#fields` is a Thymeleaf helper object for working with errors

- `#fields.hasErrors('<field name>')` checks if a field has any errors
- `#fields.errors('<field name>')` returns a list of errors for a field

```
.error {
  color: red;
}
```

Example – Displaying Errors

```
<form method="POST" data-th-action="@{/register}" data-th-object="${registration}">

    <input type="text" data-th-field="*{name}">
    <p data-th-if="${#fields.hasErrors('name')}">
      <ul>
        <li data-th-each="err: ${#fields.errors('name')}">
          <span data-th-text="${err}" class="error">
        </li>
      </ul>
    </p>

      ...



</form>
```

Return all `FieldError` for `name` field

Object Binding

```
GET /register
```

1

```
@GetMapping(path="/register")
public String getRegistration(Model model) {
  model.addAttribute("registration"
      , new Registration());
  return "registration";
}
```

2

When the HTML form is `GET`, `data-th-object` and `data-th-field` binds the model to the HTML form fields

```
<form method="POST"
    data-th-action="@{/register}"
    data-th-object="${registration}">
  <input type="text" data-th-field="*{name}">
  ...
  <button type="submit">Register</button>
</form>
```

3

When the HTML form is POST, `data-th-object` and `data-th-field` binds the HTML form fields to the model

```
@PostMapping(path="/register")
public String postRegistration(Model model
    , @Valid Registration registration, BindingResult binding) {
  ...
}
```

Difference Between `GET` and `POST`

GET

- Bookmarkable because the parameters are part of the URL
- Limited to 255 characters
- Results are cached

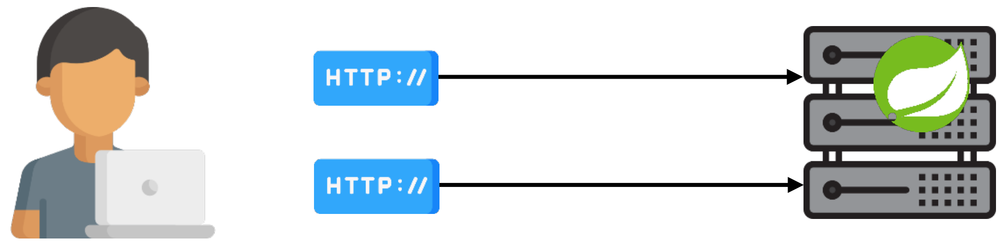
POST

- Data are carried in the body
- Need to specify the media type with `Content-Type` header
- No payload size limit

- Typical use in form submission
- GET - To retrieve some data, eg. searching for a book
- POST - To create some data, eg. RSVP a wedding

HTTP is a Stateless Protocol

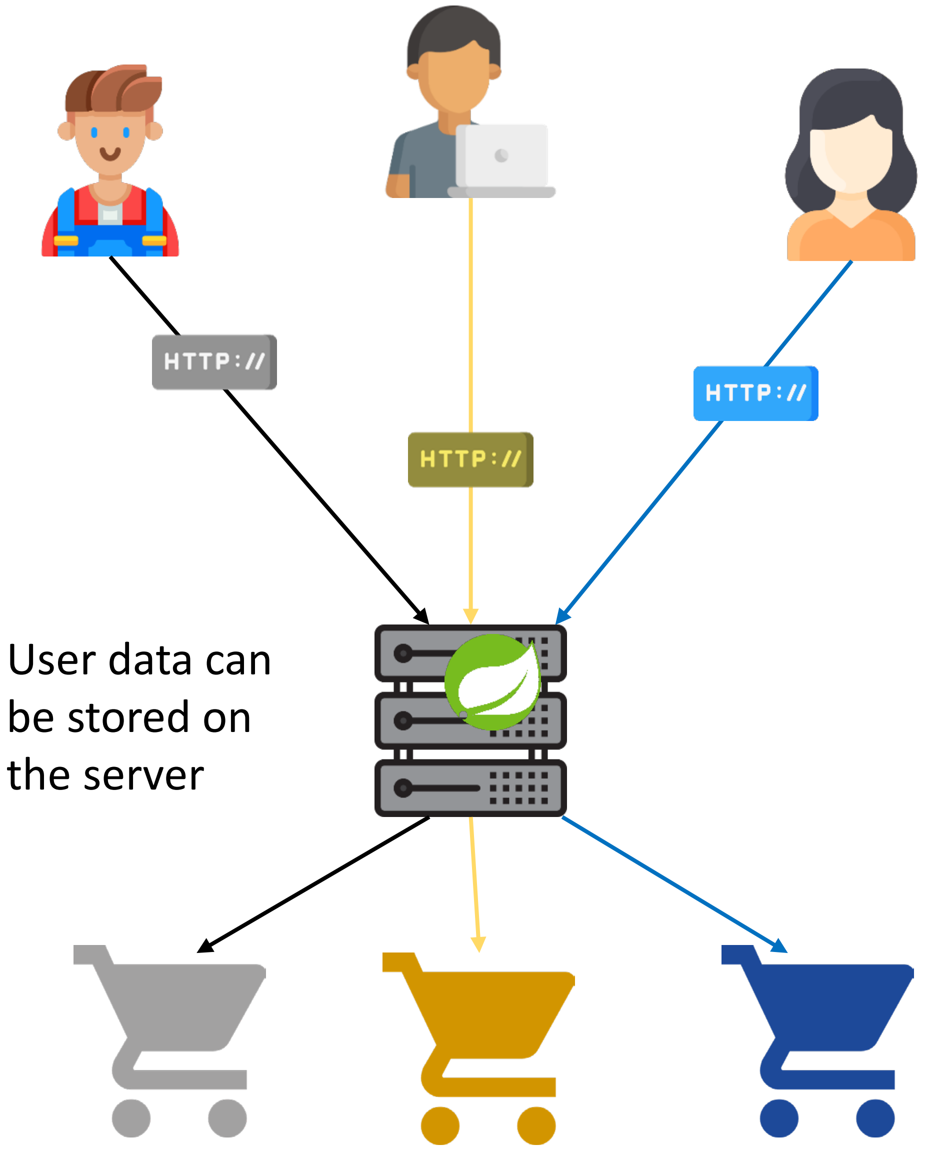
- HTTP is a stateless protocol
  - Web application cannot correlate multiple request from the same client
  - Eg. a user adding items to a shopping cart by making multiple HTTP request
  - The client has to provide some way of the server to identify the request
- Lots of business processes require a conversation between a user and the web application
  - Eg. Wizard to help a user to perform checkout



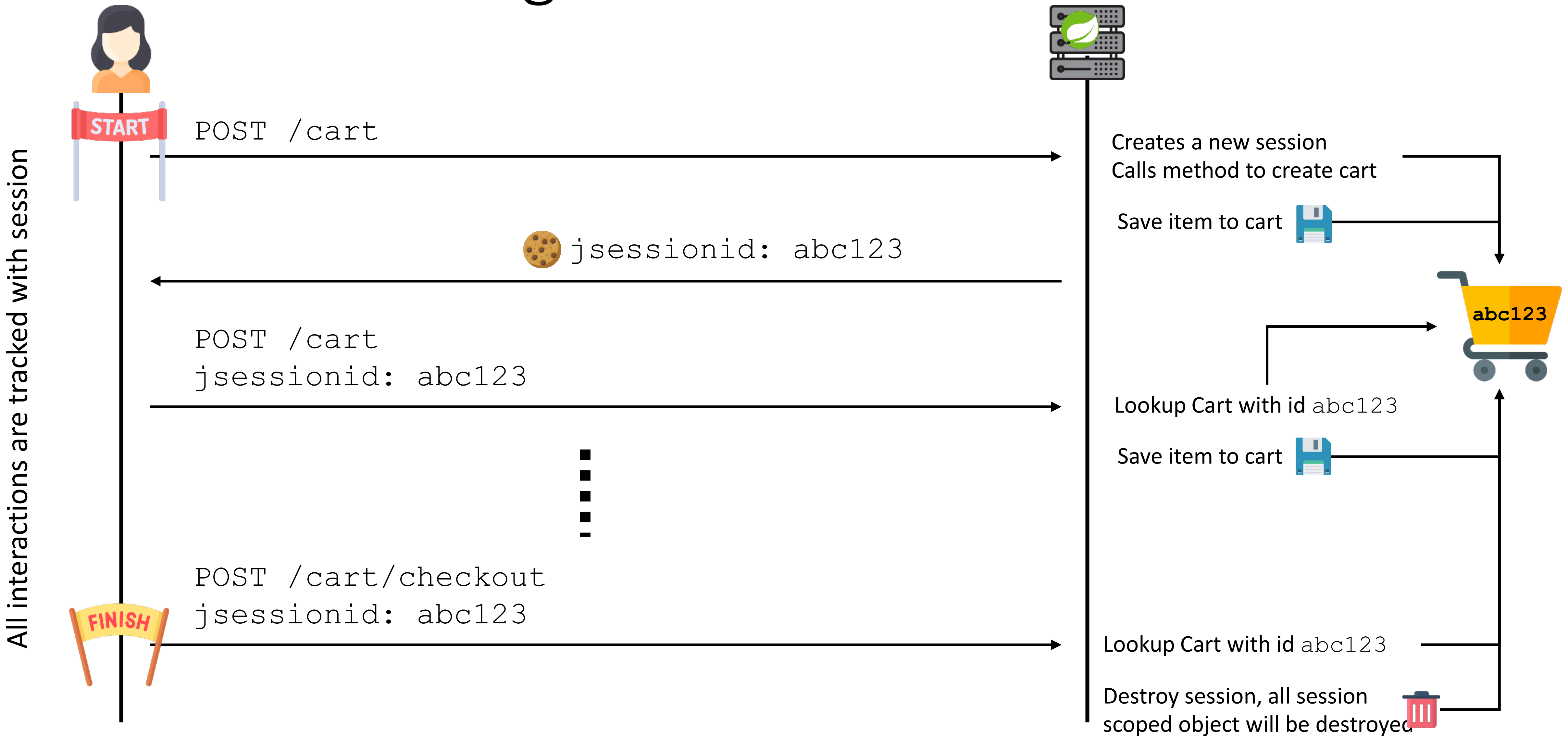
HTTP is stateless, these 2 related request cannot be correlated by SpringBoot
SpringBoot cannot safely hold data for a client because it cannot identify which piece of data belongs to with client

Session

- Session is a feature that allows the web application to identify a client
  - Using a cookie called `jsessionid`
- Using session, client specific data can be save on the server
  - These data are managed by SpringBoot
  - Client can only access their own data
- Session scoped data has a defined lifecycle
  - Created, in used, destroyed
- Uses cases includes shopping cart, multi-step business process, to do list, etc



Session Tracking


All interactions are tracked with session
START
POST /cart
Creates a new session
Calls method to create cart
Save item to cart
jsessionid: abc123
abc123
POST /cart
jsessionid: abc123
Lookup Cart with id abc123
Save item to cart
FINISH
POST /cart/checkout
jsessionid: abc123
Lookup Cart with id abc123
Destroy session, all session scoped object will be destroyed

`HttpSession` Object

- Inject into controllers
- A new session object is created when Spring establishes a new session with the client
    - Behaves like a `Map<String, Object>`
    - `HttpSession` object is associated with the `jsessionid` cookie
- Use the methods to manage session objects and also to invalidate or terminate the session
    - `HttpSession.setAttribute(“attributeName”,  <some object>)`
        - Associate the object to the `attributeName` for the duration of the session
    - `HttpSession.getAttribute(“attributeName”)`
        - Get the object with the `attributeName` from the session. Returns null if object is not in the session
    - `HttpSession.removeAttribute(“attributeName”)`
    - `HttpSession.invalidate()`
        - Destroy the session

Example - Cart and Item Model

```
public class Item {
  @NotNull(message="Item name cannot be empty")
  @Size(min=3, message="Item's name cannot be less than 3 characters")
  private String name;

  @Min(value=1, message="Minimum quantity is 1")
  @Digits(integer=5, fraction=2
      , message="Maximum quantity is 5 digits and 2 decimals")
  private Float quantity;
  // getters and setters
}


public class Cart {
  private List<Item> contents = new LinkedList<Item>();
  // getters and setters
}
```

Example - Using Sessions

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

  @GetMapping String getCart(Model model, HttpSession session) {

    Cart cart = (Cart)session.getAttribute("cart");
    if (null == cart) {
      cart = new Cart();
      session.setAttribute("cart", cart);
    }

    model.addAttribute("item", new Item());
    model.addAttribute("cart", cart);

    return "cart";
  }
}
```

Inject the session object into the controller

Try to get the cart object from the session.
If it is a new session, then the value will be null.
Initialize and create a new cart

Add the cart object to the session.
Cart will remain in the session until the session is destroyed

Example - Using Sessions

```
<form method="POST" data-th-action="@{/cart}" data-th-object="${item}">
  <input type="text" data-th-field="*{name}">

  <input type="number" min="1" step="0.01" data-th-field="*{quantity}">

  <button type="submit">Add</button>
</form>

<div data-th-unless="${#lists.isEmpty(cart.contents)}">
  <div data-th-each="it: cart.contents" data-th-object="${it}">
    <span data-th-text="*{name}">
    </span> <span data-th-text="*{quantity}"></span>
  <div>
</div>
```

Example – Using Sessions

Retrieve the item from the session. Should not be null since cart is created in the `GET /cart` handler

```
@PostMapping
public String postCart(Model model, HttpSession session
    , @Valid Item item) {

  Cart cart = (Cart)session.getAttribute("cart");

  cart.addToCart(item);

  model.addAttribute("item", new Item());
  model.addAttribute("cart", cart);

  return "cart";
}
```

Add the item from the form to the cart

Create a new Item object to be bound to the form

Destroying a Session

```
@PostMapping(path="/checkout")
public String postCheckout(Model model, HttpSession session) {

  Cart cart = (Cart)session.getAttribute("cart");
  // Calculate items in cart and perform checkout

  model.addAttribute("item", new Item());

  session.invalidate();

  return "cart";
}
```

Destroys the current session. All session scoped object will also be destroyed.
On next request, Spring Boot will create a new session

Unused

Example – Validating Form Fields

`BindingResult` contains the validation results

Syntactic validation
Validate the data capture from the form by the model

```
@PostMapping
public String postRegistration(@Valid Registration registration
      , BindingResult binding) {

   if (binding.hasErrors())
     return "registration";

   // Check for other errors
   if (!isNameUnique(registration.getName())) {
     ObjectError err = new ObjectError("globalError"
           , "%s is not available".formatted(registration.getName());
     binding.addError(err);
     return "registration";
   }

   return "thankyou";
}
```

If there are validation errors, return to the form and report the errors

Semantic validation
Add business related errors to `globalError` 'field.
The errors consist of a name and the error message.
Add the error to the `BindingResult` object.
Can add multiple errors with the same name.

Mapping Form Fields - `@ModelAttribute`

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

    @PostMapping(
        consumes = "application/x-www-form-urlencoded",
        produces = "text/html")
    public String createUser(
        @ModelAttribute User user, Model model) {

        // process the data
    }
}
```

`POST /user` should be processed by this method

For matching `Accepts` and `Content-Type`

Form fields are mapped to this object
Object must have getters and setters with matching properties

Creating a Session Object

```
public class Cart {
  private List<String> contents = new LinkedList<>();

  public void addToCart(String item) {
    contents.add(item);
  }
  public void getCart() {
    return contents;
  }
  ...
}
```

An object to be tracked by session

Tells SpringBoot that this is a configuration class. Has methods that creates object for dependency injection

```
@Configuration
public class AppConfiguration {
  @Bean
  @Scope(value=WebApplicationContext.SCOPE_SESSION
      , proxyMode=ScopedProxyMode.TARGET_CLASS)
  public Cart createCart() {

    return new Cart();
  }
}
```

Annotations to let SpringBoot know that this is a factory method to create a `Cart` session object

The `Cart` object is session scoped. Call this method to create a `Cart` object.

Using Sessions

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

  @Autowired
  private Cart cart;

  @PostMapping
  public String postCart(@RequestBody MultiValueMap<String, String> form,
      Model model) {
    String item = form.getFirst("item");
    cart.addToCart(item);
    model.addAttribute("contents", cart.getCart());
    return "cart";
  }
}
```

Dependency injection. Create an instance of `Cart` object. If this is a new session, call factory method (`@Bean`) to create it; if this is an existing session, look for the existing instance associated with this request

Add an item to the session object. SpringBoot will lookup the same instance for request coming from the same client

Mapping Form Fields - `@ModelAttribute`

```
name=fred&email=fred@gmail.com&phone=12345678
```

```
public class User {
    private String name;
    private String email;
    private String phone;
    public User() { }
    public String getUser() { ... }
    public void setUser(String u) {...}
}
```

SpringBoot instantiates `User`, injects the form fields into the object and passes it to the request handler

```
@PostMapping(...)
public String createUser(@ModelAttribute User user) {
```