12.1. Models in MVC

In the previous chapters, you learned about Thymeleaf Views, which display data and an interface for a user, and Controllers and Routing which determine what data to send the the views. This data needs to come from some source and take some shape. Cue the models.

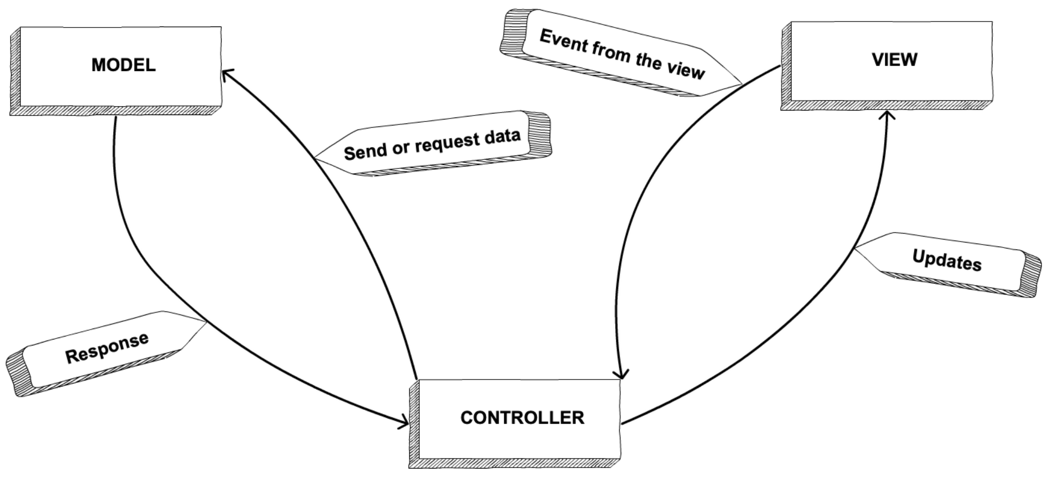
12.1.1. What is a Model?

A **model** represents the logic for accessing and storing the data used in an application. Properly constructed, they do not depend on any controllers or views, which makes models easy to reuse without modification.

Models are not the data itself, but rather the data logic. They dictate how we want to handle the data in an application-specific way. The data used in an application is often sourced from a database or an external data service. Data is typically application-agnostic and it is the work of the models we write to shape raw data into useful application information.

Consider a physical address book (a model). It stores names, addresses, phone numbers, etc. on its pages. Anyone (a controller) can pick up the book, retrieve the information, and then write some of the data on a separate piece of paper (a view). The address book model does not depend on who picks it up and enters their contacts. The book just provides organization and storage logic. On the flip side, the same person can input the same contact data into a different book. So a model takes raw information and turns it into something useful for a particular application.

12.1.2. MVC: Putting it Together



Each MVC node carries its own responsibilities.

12.1.2.1. Model

Shapes data to fit the needs of an application.

12.1.2.2. View

Displays data to the user. Via events, the user interacts with the view and updates the program data. The view communicates with the controller but not the model.

12.1.2.3. Controller

Directs the flow of information between the view and the model. It does not store the data or determine how to display it for the user. It passes information retrieved from the view to update the model. And it passes information retrieved from the model to update the view.

Tip

Need further review? Check out MVC for Noobs.

12.1.3. Model vs. Controller

One tricky part of using the MVC structure is deciding what code belongs in the model and what belongs in the controller. In general, any code that deals with data transfer or responding to user actions belongs in the controller. Code that involves retrieving data from a database or service or organizing that data belongs with the model.

In our `coding-events` application thus far, we've done all data handling inside of controller classes. However, most data manipulation should occur in model classes. So we need to make a distinction between these actions. For any manipulations that must occur regardless of a user's actions, that code belongs in the model. For changes that vary depending on a user's request (e.g. multiplying vs. adding a set of numbers), that code belongs in the controller.

Model code defines the logic for processes that the user never needs to see. These include:

1. The mechanics for storing data.
2. The mechanics for retrieving data.
3. The mechanics for organizing data.
4. Updating or manipulating the data independent of any controller or view actions.

Controller code handles *requests* made by the user. These include:

1. "Please retrieve this information from the model.",
2. "Please store this according to the rules of the model.",
3. "Please delete this item.",
4. "Please change this item.",
5. "Please display this.",
6. "Please modify this data in a novel way".

Remember, the controllers are the traffic cops, directing information from one place to another. A controller does not permanently store data. Instead, it either sends the information to the model, which uses its own code to preserve the data, or it sends the content to the view for display.

12.1.4. Check Your Understanding

Question

If we use baking as an analogy for an MVC project, which of the following items best represents a *model*?

1. The bread ingredients: flour, water, yeast, salt
2. Mixing and shaping the ingredients together
3. Baking the loaves into the final product
4. Eating the bread

Question

If we use a library as an analogy for an MVC project, which of the following items best represents a *model*?

1. The books on the shelves
2. The Dewey Decimal storage system
3. The librarians
4. The book readers

12.2. Create a Model

In the next several pages, we will be making updates to `coding-events` to demonstrate model creation, how models relate to data, and the practice of model biding. The first of these steps is to move data handling out of our controller classes and into a model class. As we discussed on the previous page, the controller class is not responsible for holding data.

In `coding-events`, we'll remove the `ArrayList` data from `EventController` and create a proper Java class to deal with event items. We'll then update our controller methods to take advantage of the new model and its properties, rather than the strings stored in the list. Lastly, because the controller is updating, the template variables it relies upon will also need to change to reflect the model properties.

12.2.1. Create a Model Class - Video

The final code from this video is in the create-model branch of `coding-events`.

12.2.2. Create a Model Class - Text

Like controllers, model classes are conventionally located in a `models` package. Structurally, model classes most closely resemble the kinds of classes we practiced making at the start of this course, before introducing Spring Boot. In other words, models are **plain old Java objects**, or **POJOs**.

To create a model to shape event data, we include a field for `name`. Of course, we'll also like at least one constructor and some getters and setters.

In `models/Event.java`:

```
public class Event {

   private String name;

   public Event(String name) {
      this.name = name;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   @Override
   public String toString() {
      return name;
   }
}
```

Now that we're working to move the data handling out from the controller classes and into a class of its own, we'll need to update the `POST` handler that creates new events. Update the `.add()` method inside of `processCreateEventForm` to add a new `Event` instance:

```
@PostMapping("create")
public String processCreateEventForm(@RequestParam String eventName) {
   events.add(new Event(eventName));
   return "redirect:";
}
```

And you'll notice, we're adding a different type of data to the `ArrayList`, so we'll have to update that too:

```
private static List<Event> events = new ArrayList<>();
```

Back in the `events/index.html` template, update the HTML to use the `Event` object's fields, rather than simply strings.

```
<td th:text="${event.name}"></td>
```

Note

The syntax `event.fieldName` runs a getter method behind the scenes in order to access a field.

12.2.3. Add a Model Property - Video

The final code from this video is in the add-property branch of `coding-events`.

12.2.4. Add a Model Property - Text

To round out the `Event` class, we'll add a `description` field to showcase what our events are all about.

Updates to `models/Event.java`:

```
public class Event {

   private String name;
   private String description;

   public Event(String name, String description) {
      this.name = name;
      this.description = description;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getDescription() {
      return description;
   }

   public void setDescription(String description) {
      this.description = description;
   }

   @Override
   public String toString() {
      return name;
   }
}
```

Now that our data is object-oriented, it's quick and easy to add a new property affiliated with an event. If after this, we decide to add a `date` or `location` field, we would simply follow the pattern established. Before, with events stored as name strings, we would have had more changes to make in order to add other information fields to the shape of the data.

Update both the `events/create.html` and `events/index.html` templates to create an event object with a description field and to display that description along with the event's name.

`events/create.html`:

```
<label>
   Description
   <input type="text" name="eventDescription"  class="form-control">
</label>
```

`events/index.html`:

```
<td th:text="${event.description}"></td>
```

Lastly, add a parameter to the `processCreateEventForm` to handle the form submission and pass the description value into the creation of the Event object.

`EventController`:

```
@PostMapping("create")
public String processCreateEventForm(@RequestParam String eventName, @RequestParam
String eventDescription) {
   events.add(new Event(eventName, eventDescription));
   return "redirect:";
}
```

12.2.5. Check Your Understanding

Question

True/False: Model code is framework independent.

1. True
2. False

Question

In `coding-events`, if we add a field to the `Event` model to record the date of the event, which of the methods in `EventController` will need to be updated?

1. `displayAllEvents`
2. `displayCreateEventForm`
3. `processCreateEventForm`
4. no controller methods need to be updated

12.3. Models and Data

In order to work with data, we need to add another element to our MVC application. Say for example, we want to do things like remove an event from our list. Well, if two events both have the same name, how might we identify which of those items to delete? We can't yet. So we need to tweak how we store event data.

In `coding-events`, we add a unique identifier field to `Events` to better handle and track distinct `Event` instances. We'll also create another model class called `EventData` to encapsulate data storage and prepare ourselves for decoupling data from controller classes.

12.3.1. Add a Unique Id - Video

The final code from this video is in the add-id branch of `coding-events`.

12.3.2. Add a Unique Id - Text

Identifying data by a user-defined string called `name` is not a sustainable or scalable method of handling data in most situations. Consider the address book example. How can we distinguish between two contact entries with the same name field? It is a frequent practice to add a **unique identifier** field (sometimes called, or even labelled, **uid**) to a class responsible for modelling data. This ensures that our address book can contain two separate entries for our contacts who have the same name as one another.

To accomplish the same data clarity with events, we'll add a few things to the event model class:

1. A private `id` field .
2. A static counter variable, `nextId`.
3. Additional constructor code that:
    1. Sets the `id` field to the `nextId` value.
    2. Increments `nextId`.
4. A getter method for the `id` field.

The result in `models/Event,java`:

```
public class Event {

   private int id;
   private static int nextId = 1;

   private String name;
   private String description;

   public Event(String name, String description) {
      this.name = name;
      this.description = description;
      this.id = nextId;
      nextId++;
   }

   public int getId() {
      return id;
   }

   // ... other getters and setters ... //

}
```

With these additions, every time a new event object is created it is assigned a unique integer to its `id` field.

12.3.3. Create a Data Layer - Video

The final code from this video is in the create-data-layer branch of `coding-events`.

12.3.4. Create a Data Layer - Text

Now that we've begun building a model, it's a good time to remind ourselves that models are not designed to be data storage containers. Rather, models are meant to shape the data stored in another location into objects that can be used in our application. As we work our way into learning about database usage and service calls, however, we'll use a Java class to store some data temporarily.

Create a new package called `data` and add a class `EventData`. Whereas `Event` is responsible for organizing user-inputted information into a Java object, `EventData` is responsible for maintaining those objects once they are created. `EventData` is itself a Java class that stores events. It contains several methods for managing and maintaining the event data that simply extend built-in HashMap methods.

The contents of `data/EventData.java`:

```
public class EventData {

   private static Map<Integer, Event> events = new HashMap<>();

   public static Collection<Event> getAll() {
      return events.values();
   }

   public static void add(Event event) {
      events.put(event.getId(), event);
   }

   public static Event getById(Integer id) {
      return events.get(id);
   }

   public static void remove(Integer id) {
      if (events.containsKey(id)) {
         events.remove(id);
      }
   }

}
```

With `EventData` managing event data, we must once again refactor `EventController` to update the items stored in `EventData`. In keeping with the objective to remove data handling from the controller, we'll remove the list of events at the top of the class. Consequently, for the `displayAllEvents` handler, we'll now use events from `EventData` in `addAttribute()`:

```
model.addAttribute("events", EventData.getAll());
```

And back to `processCreateEventForm`, we'll make use of the `.add()` method from `EventData`:

```
EventData.add(new Event(eventName, eventDescription));
```

12.3.5. Delete an Event - Video

The final code from this video is in the delete-events branch of `coding-events`.

12.3.6. Delete an Event - Text

Now that we've refined our events storage method, we are able to tackle the task of deleting an object. To delete an event object from storage, we'll grab the event's id and use that information to call the `remove` method of `EventData`. Since the delete event is user-initiated, a controller will be involved to pass the information from the user-accessible view to the data layer. So our first step with this task is to create a controller method to get a view to delete events.

Onto the end of `EventController`, add the following method:

```
@GetMapping("delete")
public String renderDeleteEventForm(Model model) {
   model.addAttribute("title", "Delete Event");
   model.addAttribute("events", EventData.getAll());
   return "events/delete";
}
```

We'll now need to create a new view for the path mapped in the method above. Add a new template, `events/delete.html`. This view will reference event id fields in order to distinguish which items the user will request to delete via checkbox inputs.

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org/">
   <head th:replace="fragments :: head"></head>
   <body class="container">

      <header th:replace="fragments :: header"></header>

      <form method="post">

         <th:block th:each="event : ${events}">
            <div class="form-group">
            <label>
                  <span th:text="${event.name}"></span>
                  <input type="checkbox" name="eventIds" th:value="${event.id}"
class="form-control">
            </label>
            </div>
         </th:block>

         <input type="submit" value="Delete Selected Events" class="btn btn-danger">
      </form>

   </body>
</html>
```

We also need a `POST` handler to take care of what to do when the delete event information is submitted by the user. We'll have this post handler redirect the user back to the home page once they have selected which event, or events, to remove from storage.

In `EventController`, add another controller method:

```
@PostMapping("delete")
public String processDeleteEventsForm(@RequestParam(required = false) int[] eventIds)
{

     if (eventIds != null) {
         for (int id : eventIds) {
             EventData.remove(id);
         }
     }

     return "redirect:";
}
```

This handler method uses the `required = false` parameter of `@RequestParam` to make this parameter optional. This allows the user to submit the form without any events selected. Once `eventIds` is optional, we must also check that it is not `null` before entering the loop.

12.3.7. Check Your Understanding

Question

In `coding-events`, which method can we call to list every event object?

1. `Events.get()`
2. `EventData.getEvery()`
3. `Event.getAll()`
4. `EventData.getAll()`

Question

In `coding-events`, breaking up the event storage from the `Event` model is an example of which object-oriented concept?

1. Inheritance
2. Polymorphism
3. Encapsulation
4. MVC design

12.4. Model-Binding

We now introduce a useful technique to auto-create model instances, called **model binding**. Model binding takes place when a whole model object is created by the Spring framework on form submission. This saves us the effort, and the code, needed to pass in each form field to a controller.

Model binding reduces the amount of code we need to write to create an object and it helps with validation (which we'll explore further in the next section). Because we use the `@ModelAttribute` annotation, Spring Boot will create an `Event` object for us when it gets the `POST` request from `/create`.

12.4.1. How to Use Model-Binding - Video

The final code from this video is in the model-binding branch of `coding-events`.

12.4.2. How to Use Model-Binding - Text

With the `Event` model in place, we can incorporate another annotation, `@ModelAttribute`. When submitting the event creation information, rather than passing in each field used to instantiate a model, we can instead pass in `@ModelAttribute Event newEvent` as a parameter of the controller method.

Revised `processCreateEventForm` in `EventController`:

```
@PostMapping("create")
public String processCreateEventForm(@ModelAttribute Event newEvent) {
   EventData.add(newEvent);
   return "redirect:";
}
```

This is the essence of model binding. The model instance is created on form submission. With only two fields needed to create an event, the value of this data binding may not be particularly apparent right now. You can imagine, though, with a larger form and class, that `@ModelAttribute` is quite an efficient annotation.

For binding to take place, we must use the model field names as the form field names. So back in the create form HTML, we update the form fields to match the event fields.

`events/create.html`:

```
<div class="form-group">
   <label>Name
      <input type="text" name="name" class="form-control">
   </label>
   <label>
      Description
      <input type="text" name="description"  class="form-control">
   </label>
</div>
```

If a form field name does NOT match up with a model field, then binding will fail for that piece of data. It is critically important to ensure these names match up.

12.4.3. Check Your Understanding

Question

Complete this sentence: Model binding …

1. requires an `@ModelAttribute` annotation.
2. helps with form validation.
3. reduces controller code.
4. is useful for all of the reasons above.

Question

In `coding-events`, we add an additional private field, `numberOfAttendees`, to the `Event` class. What other change must we make to ensure the user of our application can determine this value? (Assume we are using model binding to process form submission.)

1. Pass in a `numberOfAttendees` parameter to the form submission handler.
2. Add another input element to the create event form with a `name=numberOfAttendees` attribute.
3. Add a `getAttendees` method to `EventData`.
4. All of the above.

12.5. Exercises: Edit Model Classes

Add functionality to edit event objects in the `coding-events` application. These exercises assume that you have added all of the code from this section of the book and your application resembles the model-binding branch.

The edit form will resemble the form used to create an event.

Tip

As you work through these steps, test your code along the way! With each change you apply to your code, ask yourself what you expect to see when the application is run. You may not find that all of the steps result in observable changes, though. Use IntelliJ's debugger and read your error messages if you run into issues after applying any of the changes.

1. Create the two handler methods listed below in `EventController`. We'll add code to these in a moment, so just put the method outline in place for now.
    1. Create a method to display an edit form with this signature:

       ```
       public String displayEditForm(Model model, @PathVariable int eventId) {
          // controller code will go here
       }
       ```

    2. Create a method to process the form with this signature:

       ```
       public String processEditForm(int eventId, String name, String description)
       {
          // controller code will go here
       }
       ```

2. Add the necessary annotations to these methods for them to both live at the path `/events/edit`.
    1. Judging by the names of the handlers, which should handle `GET` requests and which should handle `POST` requests?
    2. Remember, we've configured `@RequestMapping` with a URL segment on the controller class already.
    3. You'll need to configure the route for `displayEditForm` to include the path variable `eventId`, so that paths like `/events/edit/3` will work.
3. Create an `edit.html` view template in `resources/templates/events`.
4. Copy the code from `create.html` into `edit.html`.
    1. You'll want to update the text of the submit button to reflect the edit functionality.
5. Back in the `displayEditForm` handler, round out the controller method.
    1. Use an `EventData` method to find the event object with the given `eventId`.
    2. Put the event object in the `model` with `.addAttribute()`.
    3. Return the appropriate template string.
6. Within the form fields in `edit.html`,
    1. Get the name and description from the event that was passed in via the `model` and set them as the values of the form fields.
    2. Add `th:action="@{/events/edit}"` to the `form` tag.
7. Add another input to hold the id of the event being edited. This should be hidden from the user:
8. `<input type="hidden" th:value="${event.id}" name="eventId" />`

9. Back in the `displayEditForm` handler, add a title to `model` that reads “Edit Event NAME (id=ID)” where NAME and ID are replaced by the values for the given event.
10. In `processEditForm`,
    1. Query `EventData` for the event being edited with the given id parameter.
    2. Update the name and description of the event with the appropriate model setter methods.
    3. Redirect the user to `/events` (the event listing page).
11. To access event editing, the user will need an edit option in the list of event data.
    1. In `resources/templates/events/index.html`, add a link in a new table column to edit the event:

```
<td>
   <a th:href="@{/events/edit/{id}(id=${event.id})}">Edit</a>
</td>
```

12.6. Studio: Spa User Signup

For this studio you will add functionality to allow users to sign up for your `spa-day` app.

The starter code has been modified from where you left off last class. Grab the refactored code on the [user-signup-starter branch]().

You'll notice in this branch that the name field has been removed from the service selection form. Once we implement user-signup functionality, we can use a given user's name to identify the spa client. We've also moved data into a `Client` model and out of the `SpaDayController` class.

In this studio, we'll ask you to write another model, `User`. `User` and `Client` may at first appear redundant, but in the future as you develop your spa application, you may find a scenario where a user is logging in who is not also `Client`.

12.6.1. Getting Ready

Within `spa-day`, create the following files.

1. Create a `UserController` in `org.launchcode.spaday.controllers`. Add the `@Controller` annotation, along with `@RequestMapping("user")` to configure routes into the controller.
2. Create a new folder, `user/` within `resources/templates`
3. Create `index.html` and `add.html` templates within `resources/templates/user/`
4. Create a `User` class within `org.launchcode.spaday.models`

12.6.2. Creating the Model

Your `User` class should have a few private fields with getters and setters: `username`, `email`, `password`.

12.6.3. Rendering the Add User Form

1. In the `UserController`, create a handler method `displayAddUserForm()` to render the form. This handler should correspond to the path `/user/add`, and for now, it can just return the path to the `add.html` template.

   Tip

   Don't forget to add `/user/add` to your path when you test your new features.

2. Within the `add.html` template, create a form that accepts inputs for each of the `User` class properties. Include an additional password input field to verify the password input. The form should be set up to `POST` to `/user`.
3. Be sure to set `type="password"` for the password and verify inputs, to ensure the passwords are not visible when being typed into the form. You can also set `type="email"` on the email input, which will enable some basic client-side validation. We'll tackle validation in more detail in the next studio.

12.6.4. Processing Form Submission

1. Within the `UserController`, create a handler method with this signature:

```
public String processAddUserForm(Model model, @ModelAttribute User user, String
verify) {
   // add form submission handling code here
}
```

   This will use model binding to create a new user object, `user`, and pass it into your handler method.

   Note

   You don't need to store the `User` object anywhere for this studio. We're focusing on form handling and validation in this exercise. If you want to keep track of users using the method we employed in the models lesson video, check out the Bonus Missions below.

5. Check that the `verify` parameter matches the password within the `user` object. If it does, render the `user/index.html` view template with a message that welcomes the user by username. If the passwords don't match, render the form again.

12.6.5. Refining Form Submission

1. Once registered, we want the user to access the form selecting their spa services.
   1. In `user/index.html`, add a `th:href` element to take the user back to the root path, `/`, of the app, where the `serviceSelection` template will be rendered.
2. If the form is re-rendered when a password is not verified, we should let the user know that their form was not properly submitted. Use `model.addAttribute` to add an `error` attribute, letting the user know that their passwords should match. This model attribute will need to correspond to an element in the template that will only render the error text when the passwords do not match.
3. If we send a user back to re-populate the form, it would be nice to not clear their previous submission. We won't need to save the password entries in this fashion.
   1. In the form submission handler, add the `username` and `email` fields of the submitted user as model attributes.
   2. Back in the form, add a value attribute to these form fields and make them equal to the model attributes.

12.6.6. Bonus Missions

1. Add an `id` field to `User`, along with accessor methods (with appropriate access level). Create a `UserData` class within `org.launchcode.spaday.data` that provides access to a list of users via `add`, `getAll`, and `getById`.
   1. In the `user/index.html` view, display a list of all users by username. Each username should have a link that takes you to a detail page that lists the user's username and email.
2. Add a `Date` field in `User`, and initialize it to the time the user joined (i.e. when the `User` object was created). Display the value of this property in the user detail view.