Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Notes	Weekly
Resources	Graded work
Policies	Standards
Professors & addendum	Code examples

Angular Forms Data Models

We learned in the “forms introduction” document (and specifically in this section) that:

> *…we always assume that an Angular form is backed by a data model.*
>
> *For typical scenarios, the data model can be defined in the same source code file as the component class that interacts with the form.*
>
> *The data model’s values are made available to the form when it is built and rendered, and updated by the form during user interaction and submission.*

The data model consists of instances of one or more custom-designed classes. Simple scenarios need a simple data model (as simple as a typed property, or a simple custom-designed class with a few properties), but typical scenarios need a more complex data model that consists of one or more custom-designed classes.

Similar to what we do when writing data model classes for the app in `data-classes.ts`, we can assign initial values for properties, if we wish, and/or write a constructor.

The goal is to define, in a data model, the *shape* of the data needed by the form, and/or expected from the form when it gets submitted.

This document has more information about this concept and development technique.

Why?

Why do we create a data model? Several reasons, among them:

- Data binding by definition *demands* a data model
- We are able to improve code quality and safety by defining input types
- Data can be sent to the form for rendering (e.g. for `select option` elements etc., or to support an “edit” task for an existing document)
- It enables and enforces *structure*, so that we know exactly what we’re expecting (which again improves code quality, safety, etc.)

In typical JavaScript-language form handling code, we often use loosely-defined structures, often using object initializers (aka literals).

However, in Angular apps (and in many other library- or framework-based approaches), we use data models that are defined by custom-designed classes.

What classes do we need?

We need properties for *every* bit of data going to or coming from the form.

Each property can be a simple built-in type (e.g. `string`). Or, it can be a custom-designed type, that you define in a new class.

> *As noted earlier, write the custom-designed class code in the same source code file as the component code, maybe at the bottom.*

Below, a range of examples are presented, with design and coding approaches.

1. Example - simple string
2. Example - simple data structure, get from user of the form
3. Example - simple data structure, edit scenario
4. Example - form that needs data for `select option` control(s)

1. Example - simple string

Assume that a component has a single `input` element that is used in a “search” task. In the component code, we define a simple property to hold the search text, and that’s it, we’re done:

```
searchText: string;
```

In the component template, we use two-way binding:

```
<input type="search" name="searchText" [(ngModel)]="searchText">
```

(Obviously, we'll need a few more bits here, notably a change handler or something to execute the search. However, we're focusing on the data model concept here.)

2. Example - simple data structure, get from user of the form

Assume that a component renders a "login" user interface (UI) - it has a text field for the user name and password, and a button to perform the login.

No data is sent to the form.
We will gather data from the form when it is submitted.

In the component code, we could simply use two string properties. However, let's write a custom-designed class:

```
class DataForm {
  username: string;
  password: string;
}
```

Then, declare a property. (If you wish to suppress a console error about null values before they get filled in, initialize the property in the constructor.)

```
user: DataForm;
```

The component template then binds to the property (some attributes were omitted). Notice how the binding is done:

```
<input type="text" name="username" [(ngModel)]="user.username" requir

<input type="password" name="password" [(ngModel)]="user.password" re
```

Form submission, and the data from the form

After the form is submitted, the `user` property holds the data that the user entered. What can we do with the data?

Well, if it has the exact same shape that is required by the web API, the button submit handler

method can send it along to the data manager service method as-is.

However, if the web API requires a *different* shape, perhaps requiring more properties, and/or calculated or generated values, and/or differently-named properties, then we must prepare a *new* package for the web API request, and copy over the properties from the local property. How? At least a few ways:

1. Create an instance of the class that the web API requires, manually-assign the matching property values from the local property, and add (calculate/generate) the values for the others.
2. Automate some of this by using the new-ish JavaScript object spread syntax.

A document titled JavaScript Spread and Object Mapping describes how to do #2.

3. Example - simple data structure, edit scenario

Assume that a component renders UI that enables an existing document to be edited and updated. Maybe it has a couple of fields (e.g. email address, or annual salary), and a button to perform the update.

The document's existing data MUST be sent to the form.
We will gather data from the form when it is submitted.

In the component code, we could use simple properties. However, let's write a custom-designed class:

```
class DataForm {
  _id: string;
  email: string;
  salary: number;
}
```

Then, declare a property.

```
employee: DataForm;
```

In the `ngOnInit()` method, fetch or marshal the document's data.

The component template then binds to the property (some attributes were omitted). Notice how the binding is done:

```
<input type="text" name="email" [(ngModel)]="employee.email" required
```

```
<input type="number" name="password" [(ngModel)]="employee.salary" re
```

After the form is submitted, the task is the same as described above.

4. Example - form that needs data for `select option` control(s)

Often, a form has one or more controls that enable the user to make a selection and/or choice. For example, when gathering an address, we typically offer a dropdown list (or something similar) to gather the province (Ontario, Quebec, etc.).

All such selection/choice controls require at least *two* pieces of data for each selection/choice:

1. The visible text that titles or describes the selection/chocie
2. The programmatic value that uniquely identifies the selection/choice

Sometimes, these two values can be the same. However, they're typically different. For example, a data item for the province, the visible text is `Ontario`, while the programmatic identifier value is `ON`.

Also sometimes, and typically in "edit" scenarios, we also need the current before-editing selections/choices, so that they can be rendered correctly in the UI.

How is all this handled? How do we declare the data needed for a selection/choice control (e.g. a collection/array of province data)? Do we add more properties to the custom-designed form class?

No, not for the data collection.

Instead, we do one of two tasks:

1. For **simple** scenarios (simple form, maybe with only one selection/choice control), declare (then configure) a component code property to hold the data for the selections/choices
2. For forms that are more **complex** (maybe with multiple selection/choice controls, and/or other content that must be prepared before being sent to the form), declare (then configure) a custom-designed class (to hold the data…).

However, we MUST add another property to the custom-designed form class that gathers the user's selection/choice.

For example, let's continue to work with the "employee" edit scenario from above. This time, we want to add the ability to select the employee's manager during the edit task. We need a `select option` control, like a dropdown.

Again, the document's existing data MUST be sent to the form.
We will gather data from the form when it is submitted.

Let's modify the custom-designed class:

```
class DataForm {
  _id: string;
  email: string;
  salary: number;
  manager: string;  // new, for the manager's identifier
}
```

As above, declare a property for the employee to be edited, and another property to hold the manager selections/choices:

```
employee: DataForm;

// Assume there is a class elsewhere that defines an employee
managers: Employee[];
```

In the `ngOnInit()` method, fetch or marshal the employee document's data. And, fetch or marshal the collection of managers. (This will typically be done by querying the employee collection and filtering to get only "managers".)

The component template then binds to the property (some attributes were omitted). Notice how the binding is done, and particularly the attribute values in the `select option` control:

- The two-way binding is to the property in the *employee* object
- The dropdown list items use the *managers* collection;
  each item's visible text is the *fullname* property;
  each item's programmatic identifier value is the *_id* property

```
<input type="text" name="email" [(ngModel)]="employee.email" required

<input type="number" name="password" [(ngModel)]="employee.salary" re

<select name="manager" [(ngModel)]="employee.manager">
  <option *ngFor="let m of managers" [value]="m._id_"></option>
</select>
```

After the form is submitted, the task is the same as described above.

Class names, suggestion

What should the class names be? There are no hard rules, but here's what we suggest:

- DataForm (i.e. "data for the form inputs")
- DataControls (i.e. "data for the form controls", for complex form scenarios that bundle data for MANY selection/choice controls or other purposes)

Can we define these classes - DataForm and DataControls - in other Angular components, with customized properties for each?

Yes, we can. By default, in the TypeScript compiler, file scope is enforced. Therefore, we can put these classes into different `.ts` source code files.
