Birkbeck
UNIVERSITY OF LONDON

Forms and Local Storage

Mobile Application Development

Session 6

jQuery Mobile Forms

- jQuery Mobile takes native HTML form controls and renders them as *rich controls* in a process called *auto-initialization*.
- The following controls are rendered in *rich control* format:
  - Buttons
  - Text boxes
  - Textarea
  - Checkbox and radio buttons
  - Select boxes
  - Sliders

Rich Controls (Widgets)

- Rich controls are page *widgets*.
- They are designed for touch interaction, particularly the need to cut down on typing on small screens (e.g. dates).
- Any and all interface elements in jQM are classified as page widgets (including listviews, collapsible content controls, etc.)

Form Submission

- In jQM, form submission is handled using AJAX (Not HTTP). This allows a jQM transition between the form and the results page. Example
- To ensure that a form submits as intended, we always need to specify the *action* and *method properties* on the form element.
- If no form method is specified, AJAX uses GET by default. Example.
- We can force HTTP form requests using *data-ajax="false"* on a <form> element. But, we would only really do this if we were submitting the form to an external, non-jQM, domain. Example.

jQuery Form Controls

- Because the jQuery Mobile architecture permits multiple jQM pages to be loaded into the DOM at the same time, the *id* attributes of jQM form controls should be not only unique on a given page, but also unique across all the pages in an app.
- If not, it can lead to confusion and erroneous behaviour when forms are submitted.

Form Labels

- For accessibility purposes, jQM requires that all form elements have accompanying labels. But, as labels take up valuable screen space, it is best practice to hide them using the class, *ui-hidden-accessible*.

```
<label for="username" →
class="ui-hidden-accessible">Username:</label>
```

- The label instruction can then be presented as placeholder text in the control using the HTML 5 *placeholder* attribute. Example

```
<input type="text" name="username" →
id="username"  placeholder="Username"/>
```

Field Containers

- Field containers (not to be confused with fieldsets) are a wrapper for form components that optimize the user experience for different screen sizes.
- Field containers align a control to other controls more precisely.
- They also position form labels based on a device screen width.
  - Narrower screen – label placed above control
  - Wider screen – label paced next to control
- Example

Mini sized elements

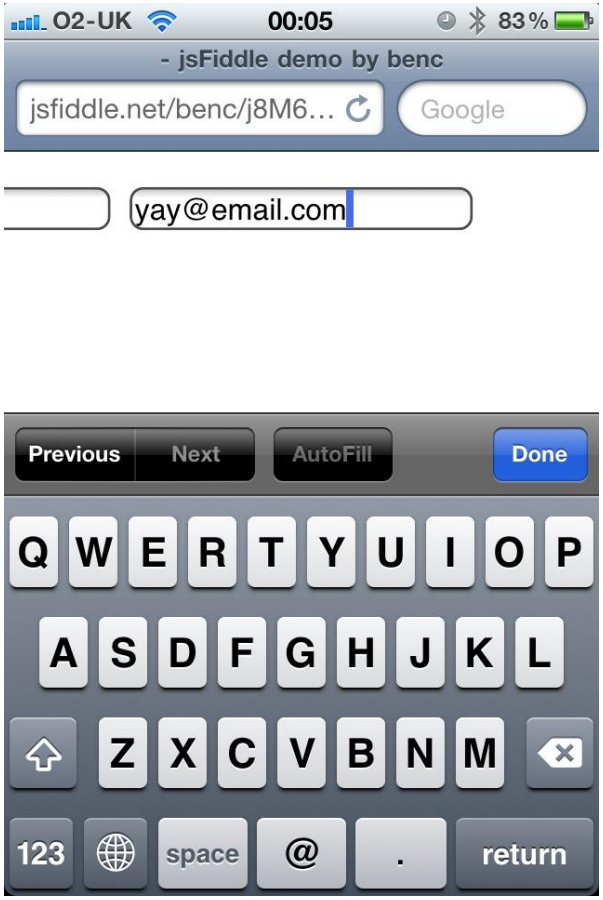
- Because forms will often be displayed in tight spaces, jQuery Mobile provides mini-sized versions of form elements.
- To use mini form elements, we add a *data-mini* attribute to the form elements in the markup.
- Even though elements with *data-mini* specified are notably smaller, they are still large enough to be touch friendly in the smallest screens.

```
<input type="submit" class="ui-btn"
data-inline="true" value="Submit" data-mini="true">
```

- Example

Text Fields

- jQuery Mobile supports all of the new features of HTML 5 form controls, including those for text input: e.g. *text*, *password*, *email*, *tel*, *url*, *search*, date, *number*.
- These input types are important additions for mobiles, as they allow a task-focused, virtual keyboards to be presented to the user.
- Example



Text Area

- In desktop applications a <textarea> element takes up several lines of the screen.
- To optimize the use of screen space in mobile devices, the <textarea> element is auto-growing.
- It starts with two lines, and adds extra lines each time a line is completed.
- Be wary, about using this control, however. It is better to minimize the amount of text a user has to type in smaller screens.
- Example

Date Field

- The jQM date field is designed so that the user should not have to type at all.
- When a user enters a date field, a date-picker widget is revealed, and the user can select the date by spinning or touching (Note that this works differently on different platforms).
- Browsers that do not support this feature will fall back to plain text input (progressive enhancement at work). Example

Radio Buttons and Checkboxes

- Radio buttons and checkboxes are larger in jQM for better finger accuracy.
- For radio or checkbox buttons to work correctly in jQuery Mobile, each button must:
  - be *type*=radio or checkbox
  - have the same *name* value (in the same controlgroup).
  - have a unique *id*
  - have a unique label associated with it
  - be grouped inside a element that has a *data-role* attribute with a value of *controlgroup*.
- To provide a label for the controlgroup, it is recommended to use a <legend> element. However, this may raise validation issues.
- Note that jQM radio buttons can be displayed either vertically or horizontally. Example

HTML 5 Enhancements – Local Storage

- Complex (and sometimes even simple) web applications often require the storage and retrieval of data.
- Bulk data generally needs to be stored server side (e.g. using PHP/MySQL).
- However, there are occasions where we may also prefer to store our data on the client side.
  - Where we want an application to work offline
  - Where data is specific to the app/user interaction (e.g. user preferences, or saved games)
- In the past, local storage was limited to Cookies.
- Now, however, HTML 5 provides the *local storage* API to allow us to store more data, and more complex data, client side. Example.

Using Local Storage

Birkbeck UNIVERSITY OF LONDON

- Advantages
  - No need for server round trip (thus faster)
  - Larger storage capacity than cookies (5MB per domain, as opposed to 4096nbytes)
  - Implemented consistently across all recent browsers
- Disadvantages
  - Security (data not encrypted)
  - Not an SQL type database that can be queried
  - Data stored only in string format
  - Not supported in older browsers

Check for Local Storage Support

Birkbeck UNIVERSITY OF LONDON

- To use local storage, we first need to make sure it is supported by the browser.

```
if (typeof(Storage) != "undefined") {
    //Process the data
} else {
    //Geolocation not supported message";
}
```

- If local storage is supported, then we store or get the data. If not, we display a geolocation not supported message to the user.

Storing Data using JSON

- Data is stored in local storage using JavaScript objects. These objects are written using the JSON.
- JSON is a lightweight data-interchange format, similar in some respects to XML.
- It is written as a collection of name/value pairs.

```
var userDetails = {fname:'Jim', →
sname:'Patel', age:23, gender:'M'};
```

Storing Data using JSON

- JavaScript objects can be also be stored within a collection (e.g. an array, vector or sequence.
- Here several objects are stored within an array.

```
var userDetails = [
  {fname:'Jo', sname:'Rex', age:23, gender:'M'},
  {fname:'Al', sname:'Lee', age:29, gender:'F'},
  {fname:'Si', sname:'Riz', age:29, gender:'F'}
];
```

Storing Data using JSON

- We use the *setItem* method of the *localStorage* object to store data in local storage.

```
localStorage.setItem(userDetails);
```

- However, local storage only allows us to store string values.
- We therefore have to first convert JSON objects to strings before we can place them in local storage (e.g. object *serialization)*.
- This is done using the JSON *stringify*() method.

```
localStorage.setItem("userDetails",→
JSON.stringify(userDetails));
```

Retrieving Data from Local Storage

- To retrieve an Object from local storage, we use the *getItem* method.

```
localStorage.getItem('userDetails')
```

- However, the data we retrieve is little use to us in string form. We thus have to convert it back to JavaScript object format.
- This time we use *JSON.parse* to complete the process.

```
 var userDetails=JSON.parse(localStorage.getItem →
('userDetails'));
```

Processing Data from Local Storage

- Once the data is converted to JSON format, we can access its component parts using standard indexing in an appropriate loop construct (in the case below, a *for in* loop).

```
function displayDetails(details) {
  var text = '';
  for(var i in details) {
      text += i + ': ' + details[i] + '<br>';
  }
 $("#result").html(text);
}
```

- Example

Processing Data from Local Storage

- Where we have a collection of JS objects stored in an array the indexing is somewhat more complicated, as essentially we are iterating through two collections simultaneously.

```
var userDetails = [
  {fname:'Jo', sname:'Rex', age:23, gender:'M'},
  {fname:'Al', sname:'Lee', age:29, gender:'F'},
  {fname:'Si', sname:'Riz', age:29, gender:'F'}
];
```

Processing Data from Local Storage

- Here, we have to index the collection, and specify the element in the collection at the same time.
- We do this using indexing and dot notation in combination. Example.

```
function displayDetails(details) {
  var person = '';
  for(var i in details) {
      person += (parseInt(i)+1) + ': '
      + details[i].firstname + " "
      + details[i].surname + ", "
      + details[i].age + ", "
      + details[i].gender + '<br>';
  }
  $("#result").html(person);
}
```