CAS CS 411
Lec 12

App Configuration

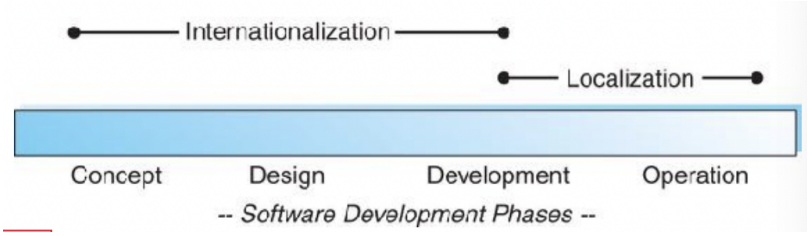
1. App and Build Configuration
    a. Applications typically must be built for a variety of environments (dev, test, production, etc)
    b. The application isn't changing, it's its environment - Databases - Server addresses - Specific hardware
    c. To solve this problem we abstract out configuration information and maintain it separately
    d. Configs are usually either XML or flat key-value lists
    e. We also need to accommodate various languages in the presentation layer…we'll look at that first
2. Localization
    a. When we build apps, we can't assume that every user will be in the same country or speak the same language
    b. We abstract out the idea of location and keep language- or location-specific information in configuration files
3. Locale
    a. A locale is a set of parameters that define preferences based on
        i. natural language
        ii. culture (often associated with a country)
    b. Preferences include spoken language, text presentation, data formats, etc.
    c. Locales are specified by combining two-letter codes
        i. language: en, fr, sp, etc.
        ii. country: US, CA, FR, etc.
    d. For example
        i. en_US English speaker, US
        ii. sp_US Spanish speaker, US
        iii. fr_CA French speaker, Canadian
        iv. fr_FR French speaker, France
4. Localization (L10n)
    a. Localization is the process of adapting an existing system for a new locale
        i. Ex: mywebapp.net was designed for use in the US by English speakers
        ii. we want to expand the target market to include more of N. America's major populations
        iii. We add:
            1. Spanish language content for locale sp_MX

        2. French language content for locale fr_CA
5. Element of Localization
    a. Date and Time
        i. Calendar: Gregorian is widely known, but lunar calendars are also in use
        ii. Date Formats
            1. MM/DD/YY, DD/MM/YY, DDMMMYYYY, etc.
            2. Month Names: January, Janvier, Enero, etc.
            3. Era: BC/AD, BCE/CE
        iii. Time Formats
            1. 12-hour, 24-hour
            2. AM/PM, πμ/μμ, etc.
    b. Colors
        i. Colors have different significance in different cultures
        ii. For example:
            1. Red: danger, luck, purity, passion
            2. Green: religion, environment
            3. White: purity, death, mourning
    c. Language
        i. Preferred language varies with locale
        ii. Language choice is dependent upon the character set being used
            1. Unicode is the universal set
        iii. Language also dictates character flow (left-right, up-down)
    d. Numbers and Measurements
        i. Decimal format variations
            1. 12,345.67 12.345,67 12 345.67
        ii. Currency symbols: $, £, ¥, €, etc.
        iii. Telephone number format
            1. (123)456-7890, 12-34-56-78-90, etc.
        iv. Measurements
            1. pound/gallon/foot/acre, kilogram/liter/meter/hectare
    e. Postal Address
        i. Postal formats vary by
            1. Placement of street number
            2. Postal code size and placement
            3. Spelling of country and city names
        ii. Example
            1. Mr. Henry Smith Alpo Automotive, Inc. 447 Main St. Yorktown, VA 55512 USA
            2. Herrn Hans Schmidt Alpo Auto GmbH Humboldt Straße 337 48147 Münster DEUTSCHLAND

f. Sorting Sequence
    i. Ordering of alphabets can vary by culture
    ii.

Germany	Sweden
Adams	Adams
Ångstrom	Wegner
Äthiopien	Voelker
Voelker	vonNeumann
vonNeumann	Ångstrom
Wegner	Äthiopien

6. Internationalization (l18n)
    a. Internationalization is the process of designing and developing an application so that it can be localized
    b. I18n is preparation for L10n
    c.



7. Internationalization with Java
    a. Java Classes:
        i. Locale: a locale
        ii. ResourceBundle: a collection of messages, images, etc. that are particular to a locale or set of locales
        iii. DateFormat: formats date and time for a locale
        iv. NumberFormat: formats numbers for a locale
        v. Collator: supports locale-sensitive sorting
8. Locale Class
    a. Locale loc = Locale.getDefault() gets the default locale for the JVM (server)
    b. Locale loc = request.getLocale() gets the locale of the client submitting an HTTP request
    c. The locale object is used to control L10n
9. Date / Time Formats
    a. Localized date display
        i. Locale loc = request.getLocale();
        Date day = Calendar.getInstance.getTime();
        DateFormat df = DateFormat.getDateInstance(style);
        String dateOut = df.format(day, loc);
        ii. Localized time display DateFormat tf = DateFormat.getTimeInstance(style);
        String timeOut = df.format(day, loc);
        style = DateFormat. [SHORT | MEDIUM | LONG | FULL]

10. Number Formats
    a. Localized number / currency formats
        i. double value;
           Locale loc = request.getLocale();
           NumberFormat nf = NumberFormat.getInstance(loc);
           String number = nf.format(value);
    b. Currency Formatting
        i. NumberFormat cf = NumberFormat.getCurrencyInstance();
11. Resource Bundle
    a. A resource bundle is a collection of name/value pairs that can define labels, prompts, messages, file names, etc., that are specific to a locale
    b. Resource bundle contents are used to localize document elements
12. Resource Bundle Definition
    a. One way to define a resource bundle is as a properties file.
    b. File name format:
       <bundle name>_<locale name>_.properties
       e.g., MessageBundle_de_DE.properties
    c. File contents:
        i. comments: # this is a comment
        ii. Name/value pairs: greeting = Hello!
13. Resource Bundle Example

```
# default English-language message bundle
greeting = Hello!
useridLabel = User ID:
passwordLabel = Password:
```

    a.
14. Using a Resource Bundle

    (assume bundle name is “MessageBundle”, package is “bundle”)

```
<%@ page import="bundle.MessageBundle" %>
  …
<%
  Locale loc = request.getLocale();
  ResourceBundle messages =
    ResourceBundle.get(“MessageBundle”, loc);
%>
<p><%= messages.getString(“greeting”) %></p>
<p><%= messages.getString(“useridLabel”) %>
  <input type="text" name="userid" /></p>
```

    a. . . .
15. For MEAN apps: back end
    a. On the back end, keep configuration items in a JSON config file
    b. Read the config during app startup
    c. Do NOT push the config file to github (trolls will quickly find and eat it)

16. severConfig.json

```
{
  "server": {
    "port": "8080",
    "name": "todoServer",
    "host": "localhost",
    "sessionSecret": "this is not a secret"
  },
  "oauth": {
    "TWITTER_CONSUMER_KEY" : "xT0JrIMSc2kA4ZDdia9DE3",
    "TWITTER_CONSUMER_SECRET": "1FXydshtl0xRF1P6voYhdd3I9UZd69P2kY0VZJpn0Px1Q"
  }
}
```

a.

```
//then in app.js…
var serverParams = require('config.json')('./config/serverConfig.json');
app.set('serverParams', serverParams.server);
. . .
//set up encryption key for sessions
app.use(session({secret: serverParams.server.sessionSecret}));

. . .
```

17. … we set up a config route…

a.

```
router.get("/config", function (req, res) {

    console.log("in config service");
    //config is on the serverParams object
    var params = app.get('serverParams');
    return res.json(params);

});
```

18. … and call it from the front end when Angular inits

a.

```
//fetchData sets up a constant, CONFIG, which is visible across the angular app
//Once CONFIG has been created and added to the angular app as a constant,
//tell angular to go ahead and start the application

//This solves a sync problem...if we just call for the config API and do not wait
//angular will continue on (that is a feature...non-blocking) and probably not see the
//config come back

//Note that this requires us to remove ng-app from the HTML frame
// in order to force bootstrapping to be manual

fetchData().then(bootstrapApplication);


function fetchData() {
    var initInjector = angular.injector(["ng"]);
    var $http = initInjector.get("$http");
    return $http.get('/todo/config').then(function (response) {
        theApp.constant('CONFIG', response.data);
    }, function (errorResponse) {
        // Handle error case...probably want to retry or display error message
    });
}
```

19. We then can use the CONFIG object on the front end

```
angular.module('app.services').factory(serviceId, ['CONFIG', '$http', todoService]);

function todoService(CONFIG, $http) {
    var port = CONFIG.port;
    var host = CONFIG.host;
    var resource = "http://" + host + ":" + port + "/todo";

    return {
        get: function (id) {
            var url = resource;

            if (typeof id !== 'undefined') {
                url += "/" + id;
            }
            return $http.get(url);
        },
        create: function (todo) {
            return $http.post(resource, todo);
        },
        update: function (todo) {
            // var id = todo._id;

            var url = resource;

            return $http.put(url, todo);
        },
        delete: function (id) {
            var url = resource + "/" + id;

            return $http.delete(url);
        }
    }
}
```

a.

20. Other configuration files
    a. In most apps, anything that can be configured is
    b. In object-oriented languages we use classes that read a configuration file and return a configuration object that contains config data
    c. In interpreted / scripted languages such as PHP, we often use tokenization (such as the Smarty template system) with localization stored in a database
21. Java configuration files
    a. In Java several classes are available, including Properties, which extends Hashtable
    b. The Properties object reads a file of key-value pairs and represents them in a POJO
    c. The properties can then be passed around to objects that need them
    d. Properties can be localization information, runtime variables such as database addresses and passwords, or whatever else can be configured
22. Environment variables
    a. Another common technique is to set up environment variables that are passed to the application
    b. Usually these are set up in a script that executes the application

c. Here's one for Node

```
#!/usr/bin/env bash
#Set up user ID and password
export USER_ID = "aSuperSecretID"
export USER_PASSWD = "3v3nMor3S3cret"
node myapp.js


//then in myapp.js. . .
//grab keys
let USERID = process.env.USER_ID;
let USERPASS = process.env.USER_PASSWD;
```

d. What are the obvious problems with this approach?
e. To solve the 'secrets in a file' problem we can set up the environment by hand on the command line
f. The simplest approach is to have the operator specify a key that will then be used to decrypt the secrets file
g. This gives you the convenience of having all the configs in a file and reduces the possibility of leaking sensitive config data
h. In this case we would NOT put the key in the script…we either use this sort of command

```
node -e 'process.env.foo = "bar"'
```

i. which hides the 'foo' variable inside the running Node process
j. Or we have the app prompt for a decryption key (or fetch it from a key service)

23. Bottom Line
    a. Most architects follow the adage:
    b. CONFIGURE EVERYTHING
    c. Yes, it makes your code more complex, BUT (as usual) the abstraction provides flexibility (and a level of security)