

Contents of Kārlis Siders' Notes

Week 1	2
Lecture 1: Intro	2
Lecture 2: CLI, Setup, HTML	3
Week 2	7
Lecture 3: Django Intro	7
Lecture 4: [Django Tutorial]	10
Week 3	10
Lecture 5: [PythonAnywhere Tutorial]	10
Lecture 6: Automated Testing	10
Week 4	10
Lecture 7: System Architectures	10
Lecture 8: Sys & Info Arch	14
3 to N Tier Architectures	14
Information Architecture	17
Week 5	19
Lecture 9: Info Architecture+	19
Lecture 9-C: Design Spec Example 1	22
Lecture 10: ER, Django Models, CSS	23
Entity-Relationship Model	23
Cascading Style Sheets	23
Lecture 10-C: Design Spec Example 2	25
Week 7	26
Lecture 11: CSS+	26
CSS	26
Lecture 12: Client-Side Environment (DOM)	29
Week 8	31
Lecture 13: Client-Side Scripting (JavaScript)	31
Lecture 14: JS+, jQuery	35
Week 9	39
Lecture 15: XML & JSON	39
Lecture 16: AJAX	43
Week 10	48
Lecture 17: Processing XML, DOM/SAX Parsing	48
Lecture 18: Messaging	50

Week 11.....	54
Lecture 19: MVC, Web App Frameworks.....	54

Week 1

Lecture 1: Intro

Assessment:

- Lab Exercises (10%) – 12 Feb
 - Rango
- Group Project (40%) – presentation 25/26 March
 - Design specification (10%) – 26 Feb
 - Project presentation (5%) – 25/26 March
 - Project submission (25%) – 26 March
- Moodle quizzes (10%)
 - Multiple choice
 - Friday mornings (10.30 am)
 -
- Exam (40%)

Web App

- Def – Distributed Information Management (DIM)
- Enables management, sharing, finding, modification, and presentation of info
 - Does so over a network in a distributed fashion
- Typically has many users, often geographically separated
- Ideally, DIMs enable users to access timely, relevant and useful info in a seamless manner

Types of Architecture

- System
- Information

Design Elements

- System arch (system focused)
 - Specifications and Requirements
 - High level system arch
 - User <-> Client <-> Middleware <-> Database
 - Entity relationship diagrams
 - Sequence diagrams
- Info arch (user focused)
 - User Personas
 - User needs matrix
 - Side design / URL design
 - Wireframes and Walkthroughs

User and Client

- The User could be human or machine, which
 - initiates contact and interacts with the client
 - ranges in skills and abilities
 - has a number of reqs that need to be satisfied
- The Client is a program sitting on a client device, which
 - sends request messages
 - accepts response messages
 - acts on the message
 - communicating to the user
 - or affecting the env in some way

Messaging

- The request message is sent from the client to the server to:
 - ask for some info
 - send info to be stored
 - from user input
 - or from device (sensor)
- The response message is sent from the server to a client to:
 - return the requested info
 - effect some change in the env

Messaging Protocols

- The request message protocol
 - usually HTTP request
 - embedding any data to be sent
- Response message protocol
 - HTTP response
 - JSON/XML

Middleware & Backends

- The Middleware/Application Server is the central component, which
 - accepts request msgs from clients
 - returns response msgs
 - coordinates the app components
- Backend/Database is typically on a separate node and
 - stores data for the app
 - provides data when needed
 - needs to be scalable and reliable
 - could be database, index, flat file

Lecture 2: CLI, Setup, HTML

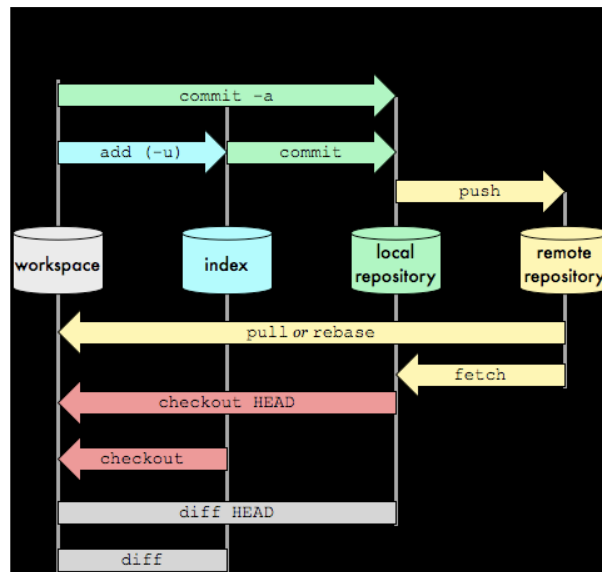
Command-Line Interfaces (CLIs)

- Control software or an OS by issuing text-based commands
- Alternative to Graphical User Interface (GUI)

- An OS might have both, ability to choose
- Can be used to perform many common tasks:
 - Navigate around directory (folder) structure
 - Create/edit files
 - Run apps
 - etc
- Common commands:
 - `dir` – list files in current directory (ls on UNIX)
 - `mkdir <name>` - create new dir
 - `cd <name>` - change directory/navigate to named destination
 - `cd ..` – move ‘up’ one level

Setting up

- Good Software Engineering practice to use:
 - version control (Git)
 - ex: CVS, Subversion (SVN), Mercurial, Git (most popular)
 - Maintains a history of a software project
 - Often stored remotely
 - Multiple users can contribute to code
 - Common practice in industry and open-source projects
 - Essential for teams but useful for individuals
 - Why?
 - Access to older (working) versions of code
 - Keeps track of different versions and releases
 - Simplifies concurrent work
 - Compare/understand changes made by others
 - Enables changes to be merged (easily)
 - Safeguards code against disaster (esp. if repo is in cloud)
 - Enables exploratory work (branching)
 - Git
 - Originally developed by Linus Torvalds (Linux)
 - Several benefits over older VCSs
 - efficient, flexible controls
 - ‘extra step’ of local repo – easier branching, encourages more commits



-
- Common commands:
 - `git clone <remote_repo>` - make a copy of the repo (once)
 - `git pull` – get the latest remote changes to local repo (merges with code files)
 - `git status` – find out state of index, changes in workspace, ...
 - `git add <filename>` - add to index (or * for everything)
 - `git commit -m "message"` – add changes to local repo
 - `git push` – uploads changes to remote repo
- Tips:
 - Always pull to work on the newest version
 - Commit early, often, then push frequently
 - Biggest hassle is dealing with merge conflicts
 - If the remote repo has changed, it is your responsibility to merge the versions
 - Communicate with team
 -

- package manager (pip)

- Software tools that automate the process of installing, upgrading, and configuring software libraries
- Tracks packages installed and their dependencies
 - If pre-requisite packages are not installed, will install them
- Help overcome nightmare of managing libraries, setting up software, replicating and environment
 - Defined: the list of packages is defined
 - Repeatable: easy to install the same set of libraries and versions
 - Managed: stored in the package manager and exportable
- Pip (Python Package Manager)
 - PyPI: Python Package Index is a repo of software for Python
 - Pip is used to install and manage packaged from PyPI
 - recursive acronym: "Pip Installs Packages"
 - Reduces development set up hassles
 - No path issues
 - Takes care of versions of libraries (recorded)

- Easy to export and share the “requirements” (set of libraries used)
 - Easy to install the same set of libraries on another machine
 - Works in conjunction with Virtual Environments
- Commands:
 - `pip install <name>[==<version>]`
 - `pip list` – show all installed packages
 - `pip freeze > requirements.txt` (puts all used packages in text file)
- virtual environment (Anaconda)
 - VE instance – environment that is configured to provide access to libraries, settings, hardware
 - Keep dependencies required by different projects in separate places
 - Don’t interfere with each other or the system
 - VE software – an app that implements, manages and controls multiple VE instances
 - Anaconda:
 - Advantages:
 - separation of package installation
 - Separation of Python versions
 - VEs can be created/switched easily
 - Commands:
 - `conda create -n <name> [python=3.8]`
 - `conda activate <name>` - enters VE (shown)
 - `conda deactivate`
 - `conda env list`
 - `conda env remove -n <name>`
- Integrated Development Environment (IDE) (PyCharm, IDLE)

Intro to HTML:

- General:
 - HyperText Mark-up Language
 - The language web browsers use to interpret what gets displayed when a web page is viewed
 - Mark-up language – set of tags which describe document content
 - Hyperlinks – connections between documents
 - HTML documents (web pages) contain HTML mark-up tags and plain text
- Tags
 - Keywords (tag names) surrounded by `<>`
 - Normally have opening and closing tags
 - Examples:
 - `<h1>` - header 1
 - Used just once
 - Defined the most important heading
 - Search engines use h1 to determine the content of web pages
 - h1-h6
 - `<p>` - paragraph tag

- browsers add space (margin) before and after each <p> element
 - They ignore own formatting – collapse whitespace
- Anchor tags – provide HTML hyperlinks
 - Syntax: link text
- Unordered list / list items
 -
 - List item one

- Div elements
 - Create sections to divide up the page in different ways when coupled with CSS
 - <div></div>
- Span elements
 - Group inline elements in a document (coupled with CSS)
 - blue
- Plain text
 - Between tags
 - The content displayed in the browser
- HTML Document Structure:
 - Nested tags
 - Starts with <html> tag
 - <head> tag contains info about the doc (title, etc)
 - <body> contains the html to be displayed
- Elements – opening tag to closing tag
- Element content – plain text between tags
- Empty elements – tags with no content, no end tag
 - ex:
 (line break)

Week 2

Lecture 3: Django Intro

High Level System Architecture

- User <> Client <> Middleware <> Database
- Work out what to build
- Decide what technologies will be used (in each category)
- Course: for Middleware, use Django as the Web Application Framework (WAF) for building the app server

Django

- “High-level Python Web framework that encourages rapid development and clean, pragmatic design”
- Claimed to be the web framework for perfectionists with deadlines
- History:

- Created by devs at a newspaper in 2003 who were keen on using Python to efficiently develop rapidly-produced content
 - Open-sourced in 2005, first major release in 2008
- Primary Focus:
 - Dynamic and database-driven websites
 - Content-bases websites
- Ex: Instagram, Spotify, The Washington Post, Dropbox, Quora, Pinterest
- Advantages:
 - Ability to divide a site or code module into logical components
 - Underpinned by the MVC/MVT design pattern
 - Providing flexibility and ease of change
 - Provides automatically generated web administration
 - Easier to manage the database
 - Provides many pre-packaged APIs for common tasks
 - Provides a template system to define HTML templates
 - Avoid code duplications
 - Subscribes to DRY principle (“Don’t Repeat Yourself”)
 - Allows extra control to define what the URL will be for a given view
 - URL requested by browser not directly accessing HTML file
 - Loose Coupling Principle
 - Allows separation of business logic from the presentation
 - Separation of Concerns
- Design Philosophy:
 - Loose Coupling
 - various layers of the framework shouldn’t “know” about details of each other
 - Don’t Repeat Yourself (DRY)
 - “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system”
 - Less Code
 - Quick Development
 - Explicit is better than implicit
 - A core Python principle
 - Consistency
- Modules:
 - Administration Interface (CRUD)
 - Create, Read, Update and Delete
 - Authentication Systems
 - Form Handling
 - Session Handling
 - Syndication Frameworks
 - RSS and Atom feeds
 - Caching
 - Internationalisation and Localisation

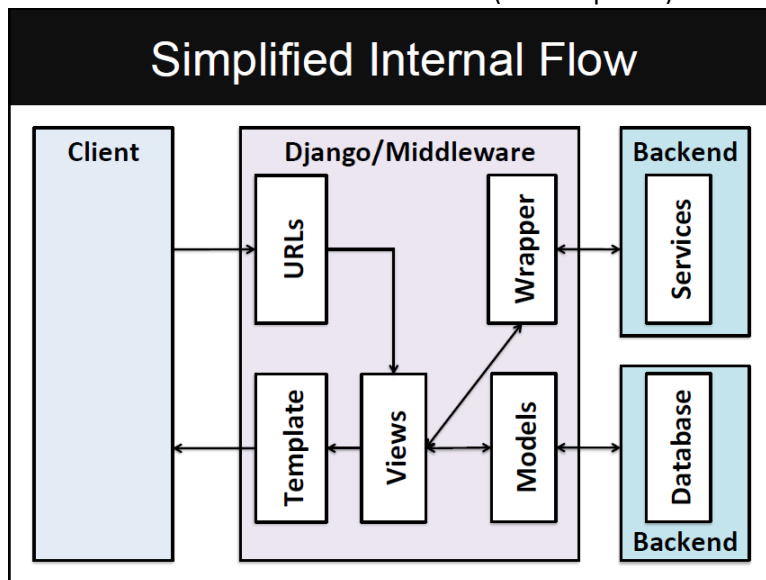
Model View Controller (MVC)

- Software architectural design pattern

- Divides software into 3 interconnected parts
- Separation of Concerns
- **Models** describe your internal data representation
- **Views** determine what the user sees
- **Controller** binds it all together (handles logic, interactions)
- No firm definitions – individual MVC designs vary significantly

MVC in Django

- **Model** describes database entities and relationships, specified as Python classes
- **Views** specified using web templating
- **Controller** is handled by
 - the Django Framework
 - URL parser, which maps URLs to views
- Definition not concrete:
 - Slight complicated because Django has ‘views’ that query/process data and might be considered part of the Controller role in MVC
 - Sometimes called MVCT or MTV (T – templates)



- Building Data Models (usually in models.py)
 - Specify the entities and relationships in the database, provide an Object Relational Mapping (ORM) to the actual database tables
 - The framework constructs the database given the models defined
- Defining Views (usually in views.py)
 - Responsible for handling and processing the specific request, collating the data from databases/external services, then selecting the template, for the response to be generated
- Controlling Flow (usually in urls.py)
 - To specify what view function should handle a particular URL (or part thereof), URL patterns are used to find matches with the URL, and to route this request to the appropriate view
 - The use of pattern matching means that different instantiations can be handled by a common pattern
- Providing Templates

- The templates mean the response format (html, xml, etc) is decoupled from the data to be presented

Lecture 4: [Django Tutorial]

PythonAnywhere

- Ability to host web apps online

Week 3

Lecture 5: [PythonAnywhere Tutorial]

Lecture 6: Automated Testing

Motivation:

- Save time
- Tests don't just identify problems, they help prevent them
- Tests make code more attractive (to use, to improve contribution)
- Tests help teams work together

Test-driven development: Write tests before writing code

Week 4

Lecture 7: System Architectures

Web Complexities

- Structure and Nature of the app
 - Hypertext structure
 - avoid user disorientation and cognitive overload
 - provide multiple paths to support users with different requirements
 - provide good superstructure, include search facilities, site map, guided tours
 - Presentation and layout
 - self-explanatory (with no user manuals for web sites)
 - aesthetically pleasing and adaptable to different contexts
 - ideally self-adapting, but somehow there must be a version of the user interface which works in each context
 - Content usage and management
 - fast, up-to-date, consistent, reliable
 - secure – especially financial transactions
 - adaptable to different contexts in terms of how much can be delivered
 - Service usage
- Usage of the app
 - **Instant** online access with **permanent** availability
 - Low tolerance for slow/hard to use sites

- A wide variety of usually anonymous users
 - Diversity in cultures and language
- A variety of **devices**
- Development process behind the app
 - By a multidisciplinary team including experts and novices
 - Development can be communal by a geographically distributed group
 - In a state of continuous development
 - Development process:
 - follows no accepted internet app development methodologies
 - must be flexible and not rigid
 - Parallel development of components (and versions) is necessary
 - Agile processes seem valuable here
 - A mixture of legacy technologies and immature technologies
 - Multiple competing products, upgrades to site often mean change of product

Types of Web App

- Static
 - Originally web apps were just collections of hand-built HTML pages
 - These required manual update and introduced the possibility of inconsistency
- Interactive
 - Forms, selection menus and radio buttons on web pages offer the possibility of selectively chosen pages generated by server-side programs
 - CGI was the first technology but has been superseded by others (ASP, JSP, PHP, Cold Fusion, etc)
- Transactional
 - Forms also offer the capture of data and database storage at the server
 - Although the data management may be separated from the internet server
- Workflow-based
 - Support for functionality expressed by a sequence of pages reflecting a business practice
 - Ex: booking flights
- Portal-oriented
 - One point of access given to multiple web sites
 - ex: Virtual Shopping malls, TripAdvisor
- Collaborative
 - Web sites which permit multiple users to share info management
 - ex: Wikipedia
- Social Web
 - Focal point for communities
 - ex: Instagram, Facebook
- Mobile and Ubiquitous
 - “Web” apps increasingly provide access by small, mobile and non-visual devices (i.e., phones) as well as data capture by sensors

System Architectures

- Every component of a system must be designed

- The architecture of the system shows the blueprint or plans of how each component fits together
- Architectural diagrams can be at various levels:
 - Data structure / Algorithm / Object level
 - Component / Library level
 - Application level
- Various diagrams are needed

Monolithic Programs

- The main tasks that any app must support:
 - UI management
 - implementation of algorithms – business logic
 - info manipulation
 - data storage
- A monolithic program in Java does all:
 - UI with Swing/JavaFX
 - algorithms in methods
 - info manipulation in methods (sorting, etc)
 - data storage with File IO

Single Tier Architecture

- User <-> UI; Methods; Data Structures; File IO

Tiered Architectures:

- Structure – different aspects separated into different levels
- Advantages:
 - Each tier can be coded separately without one programmer having to deal with everything
 - Different tiers can be distributed over the network -> efficiency++
- Effective interaction between tiers necessary
 - Well-defined interface between adjacent tiers
 - ex: Internet protocols, database connection software

Two-Tier Architectures

- Data Management
 - Handling and dealing with large amount of data is required by many apps
 - Access to a shared repo of data is highly beneficial to facilitate info flows between actors
 - Storing, retrieving, modifying, securing is common to many apps
 - Database Systems and Info Retrieval Systems provide ways to manage this data and info (efficiently)
 - Separate the concerns
 - Let Database System handle data management
 - Use a Client app to interact with the database
 - The client acts like a database user sending in queries and updates and making use of the result

- Simple technique: code SQL statements as strings inside the program
- Client-Server Architecture
 - Fat Client: Standard architecture for getting an app (e.g., Java-based) to work on top of a database
 - User <-> UI, Methods, Functionality –SQL(+Data)--> Database
 - User <-> UI, Methods, Functionality <--(Data+)ACK—Database
 - Thin Client: Standard architecture for serving static content on the web
 - User <-> Client/Browser –URL (request)--> Web server with static HTML page
 - User <-> Client/Browser<--HTML (response)—Web server with static HTML page
- Distinguish:
 - client processes
 - Control the app logic and UI
 - server processes
 - Supply resources (data) to clients
 - Issue:
 - Load on the client
 - Might tie the client software to the database software -> changes in each affect the other

Static Web Apps

- A set of hyper-linked HTML files
- Each HTML describes 1 page of the website
- Each page includes 1+ links to other pages
- If a user clicks a link or enters a URL, then a request is sent to the web server identifying the next page to load

Style and Substance

- Initially, the HTML files contained all the info on how to render the page
 - The text/data (html) and the style (css)
- With lots of pages there is lots of overhead in creating and maintaining the site
- Solution: Separate concerns
 - Use external Cascading Style Sheets which act upon the different elements in the html
 - The web server/app returns an html file (content) and a css file (style)

Two-Tier Architecture with **Layers**

- Extended Thin Client: Layers with the client tier to handle content and look/feel
- **Content layer** represents/stores the data
- **Presentation layer** renders data

Repetition of Structure

- Use templates and decompose repeated elements (separate concerns)
 - Better to have a program which generates pages by merging a template with the specific page data

- Appropriate data depends on URL and its parameters
- Enables provision of dynamic content
 - (requires a more sophisticated architecture)

Handling User Interaction

- HTTP provides methods for sending data entered by user (POST) and receiving requests (GET)

Two-Tier with Layers & Server

- Extended Thin Client and Extended Server: Layers within client handle content and look/feel, server houses web server and app server
- Web Server handles incoming requests and sends outgoing responses, routing them to/from the correct app server
- App Server is typically a script that dynamically generates a response given a request

Lecture 8: Sys & Info Arch

3 to N Tier Architectures

- An improvement is to separate the app into 3 (or N-) tiers:
 - presentation
 - processes deal exclusively with the UI – this might be through the use of a browser or other user agent
 - application
 - processes deal with the logic of the app, queries, calculations, etc – there is either one (3-tier) or more (N-tier) of such processes
 - data source
 - processes supply the data from a database (binary) or a file (XML, etc)
- Three Tier Architecture
 - The client handles the interaction with the user, the middleware handles the app logic, and the database stores the data
 - User <-> Client <-> Middleware <-> Database
 - Database: persists and manages the data associated with the app
 - Separation of Concerns: This arch further separates out responsibilities (presentation, logic, data)
 - Middleware:
 - Usually there's a webserver, an app server, and potential media servers
 - Webserver: handles incoming requests, directing them to the appropriate server
 - Servers on same/different machines
- N-Tier
 - Load Balancing
 - 2 approaches:
 - Using **Domain Name Server** such that when a URL is resolved, it rotates through a series of IP addresses that route the message to that machine
 - Using a **Load Balancing Server**, which farms out the requests to available machines

- Machines in the farm inform the LBS of their load

Tiered Architecture Benefits

- Separation of concerns
- Encapsulates complexity
 - Tiers can be broken down into layers/sub-tiers
- Distribution across several machines
 - Flexibility
 - More security (clients don't interact directly with the database)
- Replication across several machines
 - Scalability

System Architecture Diagrams

- Top-Down Design
 - Starting from a high level design is useful because:
 - helps describe system at a level which makes the **goals, scope, and responsibilities** clear
 - abstraction provides a tool for communicating design
 - permits specific technologies to be chosen late/changed
 - easier maintenance and improves reusability
 - Pros and Cons:
 - Separates low level work from higher level abstractions
 - Leads to a modular design
 - Development can be self-contained (tiered)
 - Emphasises planning and system understanding
 - Coding is late, and Testing is even later
 - Skeleton code can show how everything integrates
- Bottom-Up Design
 - How it works:
 - Piecing together components to give rise to more complex systems, thus making the original elements subsystems of the emergent system
 - Most specific and basic individual components of the system are first developed
 - These elements are then linked together to form larger subsystems, which then are linked (sometimes in many levels) until a complete top-level system is formed
 - This strategy often resembles a “seed” model, by which the beginnings are small but eventually grow in complexity and completeness
 - Pros and Cons:
 - Coding begins early and Testing can be performed early
 - Requires excellent intuitions to determine functionality of modules
 - Low level design decisions can have major impact on solutions
 - Risks integration problems – how do components link together
 - Often used to add new modules to existing system

- Diagram and Design
 - Architects need to communicate to devs how the app and its components fit together and who is responsible for what
 - The designs also serve as a communication tool with the client
 - It is important to be able to draw and read such diagrams, especially when projects become large and complex

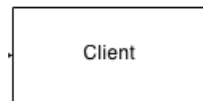
- Notations in Modified Dataflow Language:

- User (customer)



- instigates and interacts with the services or apps provided
- Types:
 - end users, admins, devs, other systems, etc

- Client (interface)



- Varies greatly:
 - web browser on PS, tablet, mobile, etc
 - an API for other systems, agents, devs, etc
 - devices and robots
 - sensors

- Middleware



- Houses many components:
 - Domain Name Servers, Load Balancing Servers, Web Servers, App Servers, Caching Servers
 - Usually, first 3 are predefined or configured using standard software
 - App Server deals with what needs to be developed
- Often represented as a single component that brokers requests between the client and database
 - Encapsulates several other components

- Database



- DB server is usually employed to handle data management side of apps (Postgres, SQLServer, MySQL)
- While system is usually already in existence, needs to be configured
 - database tables have to be defined and populated
 - ER diagrams specify this part more precisely

- Logs and External Services

- Logs represent data sinks



-
- The app outputs data but does not read it back, directly
- External Services represent apps and services that are used by the app



-
- They provide an API or interface of some kind that can be used to interact with the service
- Technology and Devices
 - For each box, the technology/device can be specified, e.g.:
 - Client: mobile Web browser, HTML/CSS/JS
 - Middleware: Apache Web Server, with App Server built by Django
 - Database: MySQL Database Server
- Data flows
 - Arrows for flow of info
 - Direction denotes direction of communication
 - Most communications are both ways (request, response)
 - Shows relation of entities

Information Architecture

General:

- Structural design of shared info environments
- Combination of organisation, labelling, search and navigation systems within web sites and intranets
- Art and science of shaping info products and experiences to support usability and findability
- An emerging discipline and community of practice, focused on bringing principles of design and architecture to the digital landscape

Communication Chasm

- Key Problem:
 - Users have particular needs, goals, aims and objectives
 - audience types
 - The organisation may/may not provide answers, suggestions, solutions

Context, Content & Users

- Context
 - Business Goals
 - Resources
 - Constraints
 - Technology
 - Politics
 - Culture

- Content
 - Document/Data Types
 - Volume
 - Existing structure
- Users
 - Audience
 - Experience
 - Needs
 - Info seeking behaviour

Info Arch importance:

- Cost of **finding**
 - time, frustration
- Cost of **not finding**
 - bad decisions, alternate channels
- Cost of **construction**
 - staff, technology, planning, bugs
- Cost of **maintenance**
 - content management, redesigns
- Cost of **training**
 - employees, turnover
- Value of **brand**
 - identity, reputation, trust

Components

- Findability
 - Info organisation
 - Navigation structures, Taxonomy, Content Search
 - Usability
 - Interaction Design
 - Industry Best Practices, W3C standards Accessibility
 - Understandability
 - Info Design
 - Metadata, Controlled Vocabulary, Labelling

Top-Down Design

- Designing for when a user arrives at the main page of the site
- Typical questions
 - Where am I?
 - How do I search for what I want?
 - How do I get around?
 - What is useful, important, unique about this site?
 - What's available, what's happening?
 - How can I get help (contact a human)?

Bottom-Up Design

- Caters for when the user lands somewhere in the site

- Typically, via search engine
- Typical questions:
 - Where am I?
 - What's here?
 - What else is here?
 - Where can I go from here?

Info Systems:

- Info Retrieval Systems
 - Users query the search engine via the search interface
 - Results are ranked and returned
- Navigation Systems
 - Global Navigation
 - main, top
 - Local Navigation
 - usually at side
 - Contextual Navigation
- Semantic Word Networks
 - Related, Synonyms, Acronyms
 - Broader, Narrower, Related

Week 5

Lecture 9: Info Architecture+

Info architect (**CONCEPTUAL**):

- assists business analysts **identify user-based requirements**
- is responsible for **how users interpret and interact with info**
 - Allows visual designers to concentrate on visual design elements and programmers to concentrate on code
- investigates **customers and their needs**, factors **business strategy and technology resources** into **solutions**
 - Done long before programming begins
 - Can minimise loss of business and wasting of resources
 - Can increase and maintain revenue from customers

Sys architect (**LOGICAL**):

- establishes structure of system
 - specifies essential core design features and elements and provides the framework for all that follows
 - provides the architect's view of the users' vision for what the system needs to be and do

- strives to maintain the integrity of the vision as it evolves during the detail design and implementation

User-Driven Approach

- Fundamental questions for IA:
 - Who is the user?
 - What do they need?
 - What will they see?
 - How will they interact with the system?
 - How will they get value from the system, accomplish their goals/tasks, etc?

IA deliverables (**TEAM PROJECT**)

- Tools and techniques to answer questions:
 - User Personas
 - User archetypes to help guide decisions about product features, navigation, interactions, and visual design
 - Demographics
 - Psychographics
 - Environmental
 - Identifying and prioritising their needs
 - User Needs Matrix
 - A document to capture user needs of various site users and prioritise them accordingly
 - Ensures that all user requirements are captured
 - Helps prioritise user needs => prioritise page elements
 - Creates a snapshot view of the user ecosystem
 - Mocking up Wireframes
 - Depicts how an individual page or template will look from an architectural perspective. They are the intersection of the site's IA and its visual and info design
 - Saves a lot of time
 - Ideas can be presented without coding
 - Helps validate page elements and structure with the user
 - Drawing:
 - Start by sketching
 - Focus on communication
 - Keep it simple, abstract, but representative
 - Number/label components
 - Explain functionality of each component
 - Map features to sys specs/requirements
 - Document and record them
 - See how design evolves, shows what's been considered
 - Use common elements
 - Get feedback on design
 - Iterate and improve

- Use a good prototyping tool to formalise design
 - draw.io – good and free
 - Heavyweight alt: Figma
- Use real content – show clients what to expect
- Develop a site map – shows context, plans navigations
 - Blueprints that show how the site is going to be organised
 - Provides high level snapshot view and relationship between pages of the site
 - Helps with restructuring the deep hierarchies
- Consider using grid layout
 - Determine layout of the main components with boxes
 - Add info using text size/highlighting to differentiate between level/importance of info
- Keep team in mind – what can be successfully developed?
 - Showing the sequence of interaction through Walkthroughs

URL Design

- Translating the site map into a coherent and logical URL design is an important part of site design
- Components:
 - Scheme (//)
 - http:, https:, ftp:, mailto:
 - Domain
 - www.netmagazine.com
 - www.w3.org
 - farukat.es
 - Path (/)
 - profile/username
 - /about
 - Query string (?)
 - ?=search
 - ?sort=alphabetical
 - Fragment identifier (#)
 - #footnote-1
- Tips
 - URLs should be obvious and inferable
 - e.g. bb.co.uk/sport/football
 - Keywords where possible
 - Shorter – better
 - Avoid too much depth
 - Use lowercase characters, avoid special chars
 - Use static URLs when you can
 - Lets users revisit info/page
 - Means that crawlers can index the content
- HTTP vs HTTPS
 - HTTPS – extension of HTTP for secure communication (S – secure)
 - Communication protocol encrypted using TLS or SSL

- Protects against MITM attacks
- Historically, HTTPS connections were primarily used for payment transactions on WWW
- Uses long-term public and private keys to generate a short-term sessions key, which is then used to encrypt the data flow between client and server
- Especially important over insecure networks such as public Wi-Fi access points

Usability Testing

- Preparing materials
 - Task sheets
 - Feedback questionnaires
- Conducting tests
- Reporting results
- Benefits:
 - Saves unnecessary effort and time
 - Helps convince clients

Lecture 9-C: Design Spec Example 1

Team D:

- Requirements
 - Not clear whether user has to be logged in for activities
- User Personas
 - Zebediah – details don't quite give a sense of what the user wants
 - Maria – too few details
- Wire Frames
 - Index
 - Good that it's consistent with requirements (beard categories)
 - Should use real images as examples, not "lorem ipsum"
 - Picture with Comment Section
 - No. of ratings not that relevant, score (+/-) should be there
 - No place to comment
 - My Uploads
 - Good consistency
- System Arch
 - Inconsistent with "Google Search"
- ER Diagram
 - **Does not use Compressed Chen notation**
- Site Map
 - Partially consistent categories (no "Trending")
- Site URLs
 - Sensible

Lecture 10: ER, Django Models, CSS

Entity-Relationship Model

- Provides an abstract representation of the data and how they are related to each other
 - Developed by Peter Chen in 1976
- Lend themselves to being implemented in a database
- 3 Main Components:
 - Entities
 - Relationships
 - Attributes
- Notations
 - Ex: Chen, Bachman, Barker, Martin, etc
 - Each propose different ways to draw the ER model
 - **TEAM PROJECT:** Compressed Chen Notation
 - Compressed Chen:
 - Rectangles - Entities
 - Diamonds - Relationships
 - 1, N or M – cardinality
 - N/M – many
 - Different notations use crows feet, etc
 - Chen vs Compressed Chen
 - Chen
 - Shows attributes as circles/ellipses
 - Disadvantage: Too many
 - Compressed Chen:
 - Only put entities and relationships in the diagram and then list the attributes separately
 - Neater
- Converting to Django Models
 - In Django, every model is automatically assigned an id
 - To create relationships between models, refer to the **model**, not id
 - In Django, primary keys are normally auto-assigned integers, but sometimes the primary key is something else (Restaurant's pk being Place)
 - Model metadata is optional (ordering options, human-readable singular/plural names, etc)
 - **TEAM PROJECT:** examples of different relationships in slides

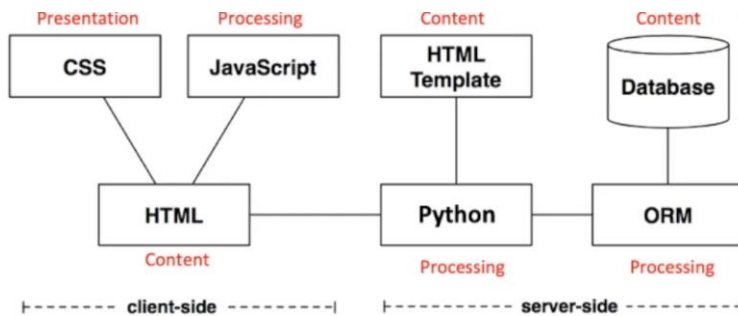
Cascading Style Sheets

Web Development Technologies

- Whilst developing web apps, the **minimums useful set of technologies** for a sufficiently complex app is 5
 - **Server-side Language (PHP, Ruby, Python, Java, etc)**
 - **Data language (SQL)**
 - **Client-side Language (JavaScript)**
 - **Content Mark-up Language (XHTML)**

- **Style Mark-up Language (CSS)**
- Mixing of technologies is difficult to maintain, and codebase is especially fragile to change

Separation of Concerns



- Separating the Presentation, Content, and Processing on the client side

Style on the Web

- Decisions of Style: most aspects about any element of a web page can be controlled (position, colour, size, font, etc)
- Decisions can be made in multiple ways:
 - Using CSS with XHTML
 - Describing page in XML and then using XSL to generate formatted XHTML
 - using CSS with XML

CSS

- Stylesheets describe the rendering of HTML elements
 - Specify stylistic aspects of **individual elements** or **all elements** of a particular **kind**
 - CSS consists of a set of **formatting rules**, which are specified like this:
 - selector {
 - property1: value1;
 - property2: value2;
 - ...
 - **Selector** indicates element (or set of elements) (h3)
 - * - all
 - # with id
 - **Property** refers to stylistic aspect (color, size)
 - **Value** is the specific configuration (yellow, 18px)
 - Units
 - Affect the colours, distances and sizes of all properties
 - Numbers
 - Integers/real
 - Percentages
 - Real number followed by %
 - Relative to some other number

- Ex: font-size: 90% of default/inherited
- Colour
 - Name ('red'), functional RGB, hexadecimal. Called **color**
- Length
 - Absolute
 - Inches (in), centimetres (cm), millimetres (mm), points (pt: 72pt = 1 in), picas (pc: 1pc = 12 pt)
 - Relative
 - em – relative to given font size
 - if font-size = 14pt, 1em = 14pt
 - ex – relative to size of a lowercase x in font size
 - px – **related** to the size of a pixel on the device
 - **Recommended**

Using CSS with (X)HTML

- Inline CSS Specification
 - **Inline** – style info is added directly to particular element using its **style** attribute
 - CSS syntax is used with the **style** attribute in an HTML tag
 - `<h3 style="color: yellow; font-size: 18px">`
 - Affects only this element
 - Useful to **override** existing style, but **breaks separation of content and presentation**
- Embedded CSS Specification
 - **Embedded**: Style rules can be specified in the <head> section of the document
 - Rules applied to entire doc
- External CSS Specification – BEST
 - **External**: In a separate document which could be shared by several pages. File extension - .css
 - <head>


```
<link rel="stylesheet" href="master.css" type="text/css">
```
 - Generally best method in terms of:
 - separation of concerns
 - maintenance
 - performance

Lecture 10-C: Design Spec Example 2

Team E:

- Requirements
 - Good – specific, distinguishes logged in and logged out users
 - Could be even clearer about logging in

- User Personas
 - Christine – needs more info (demographics, needs)
 - Adam – good
- Site Map
 - Not much shown, missing stuff
- ER Diagram
 - Wrong relationships (one to one between owner and bar)
 - Missing values for average rating
 - Should use 1 for “one” and “M”/”N” for “many”
 - Missing values for users giving ratings (price/atmosphere)
- URLs
 - /login/ as a parent dir – no
- Wire Frames
 - Home Page
 - Search should be in corner
 - Sign Up and Log In in weird places
 - Review Page
 - Sign Up and Log In inconsistent places
 - Good – shows real images
 - Make Review
 - Bar Overviews

Week 7

Lecture 11: CSS+

CSS

Specialisation of Presentation

- Class and ID selectors can be used for finer content
- Involves more planning/effort with document markup
 - Can result in better UE
 - Very important for manipulating elements in JavaScript
 - Effort pays off when using libraries like jQuery
- Class selectors work on a set of specified elements through the **class attribute**
 - Allow styling items with the same (X)HTML element differently
 - HTML – **class=“something”**
 - CSS - **.something**
- IDs provide a way to stylise unique elements through the **id attribute**
 - Similar to class, but define a special case for an element
 - Meant to be unique and only used once
 - Browsers don’t enforce uniqueness
 - HTML – **id=“smth”**
 - CSS - **#smth**

- Descendant Selectors

- Elements that are descended from a partial element are styled according to the rule of the descendant selector
 - The rules will be applied to a set of elements in 1 context, but not in another

```
<style>
  p em { color: red; font-weight: bold; }
</style>
<body>
  <p>this will be the default colour
  <em>this will be red, bold and italics</em>
  back to the default colour</p>
</body>
```

- Restricted Class and ID Selectors

All h2 elements within the class red should be coloured red

```
<style> .red h2 { color : red; } </style>
<body> <div class="red"><h2>I am red</h2> </div> </body>
```

All h2 elements whose class is red should be coloured red

```
<style> h2.red { color : red; } </style>
<body> <h2 class="red">I am red</h2> </body>
```

All h2 elements within an element with ID red should be coloured red

```
<style> #red h2 { color : red; } </style>
<body> <div id="red"><h2>I am red</h2> </div> </body>
```

Inheritance of Style

- Styles are applied not only to a specified element, but also to its descendants

Specificity of Style

- Sometimes multiple rules apply to the same element
- CSS uses a weighting scheme to ensure there are predictable outcomes with conflicts of style
 - For every ID attribute value given in the selector, add 1 0 0
 - For every class attribute value/attribute selection/pseudo-class given in the selection, add 0 1 0
 - For every element/pseudo-element given in the selector, add 0 0 1
- Order lexicographically (1 0 0 beats 0 5 5)

```
h1 {color: red;}           /* specificity = 0,0,1 */
body h1 {color: green;}    /* specificity = 0,0,2 */
#content h2 {color: silver;} /* specificity = 1,0,1 */
h2.grape {color: purple;}  /* specificity = 0,1,1 */
```

- Inline CSS overrides everything (effectively 1 0 0 0)

Cascade

- Sometimes there's still conflict (when weight is the same)
- CSS is based on a method of causing **styles to cascade together**, which is made possible by combining inheritance, specificity, and order

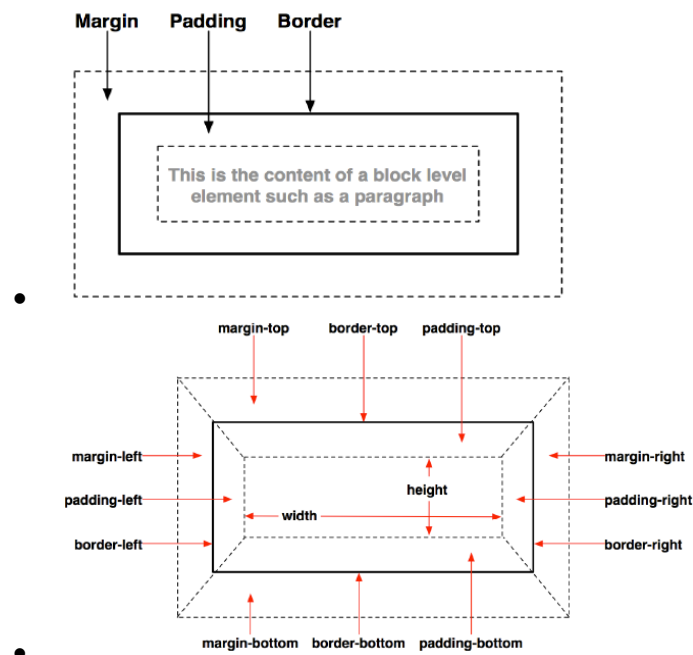
- The purpose of **cascading** is to find one winning rule among a set of rules that apply to a given element
- Rules
 - Find all rules that contain a selector that matches a given element
 - Sort all declarations by explicit weight applying to the element
 - Those rules marked “**! important**” are given higher weight
 - Sort all declarations by specificity applying to a given element
 - Those elements with higher specificity have more weight than those with lower
 - Sort all declaration by order applying to a given element
 - The later a declaration appears in the sheet/doc, the more weight it is given
 - Declarations that appear in an imported style sheet are considered to come before all declarations in the sheet that imports them

Page Layout

- Layout – columns, navigation bars, headers, footers, tables
- Preferred solution – divide a page into a collection of <div> elements
- Floating
 - CSS floating properties allow elements to **float** horizontally
 - Elements can be floated **left** and **right** (but not up and down)
 - Elements after the floating element will flow around. If screen size changes, elements will move down
- Positioning
 - **position**
 - Elements can be positioned using: **top, bottom, left, right**
 - Ways to position: **static** (default), **fixed, relative, absolute**
 - **static**
 - Always positioned according to the normal flow of the page
 - Static positioned elements are not affected by top/bottom/left/right properties
 - **fixed** – relative to browser window
 - Doesn’t matter how much scrolling/resizing happens
 - Might overlap with other content
 - **relative** – relative to its normal position
 - **absolute** – relative to its parent element
 - relative to the *first parent element that has a position other than static*
- **clear**
 - **none** (default)/**left/right/both**
 - Moves clear of floating elements

Box Model

- Every element generates 1+ rectangular **elemental boxes** which houses the content. The element box is surrounded by optional amounts of **padding, borders, and margins**



Benefits of CSS

- Method of **separating a document's structure and content from its presentation**
- Allows for a much **richer document appearance** than HTML alone
- Can save time as the appearance of the entire document can be **created and changed in just one place**
- Can improve load times as it **compactly stores the presentation concerns** of a document in one place instead of being repeated throughout the document

Summary

- Separation of concerns is a good principle to adopt
 - Simple examples usually don't benefit
 - Effort is worth it as complexity increases
- CSS is a powerful method of specifying the style of web pages
 - Separates presentation from structure and content

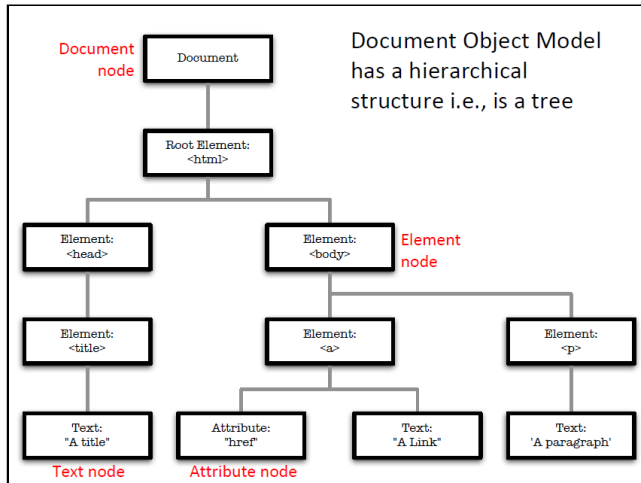
Lecture 12: Client-Side Environment (DOM)

Document Object Model

- "The DOM is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page."
- Nodes
 - In the HTML DOM, everything is a node
 - The document is a **document node**
 - All HTML elements are **element nodes**

- Can have **child nodes** of type element/attribute/text/comment nodes
 - All HTML attributes are **attribute nodes**
 - Text inside HTML elements re **text nodes**
 - Comments are **comment nodes**

- A **NodeList** object represents a list of nodes



- Traversal
 - Parent Node, First Child, Last Child, nextSibling, previousSibling
- Element Properties & Methods (JavaScript)
 - Properties – values you can get/set, like changing the content of an HTML element
 - someEl.**innerHTML** – text value
 - someEl.**nodeName**
 - someEl.**nodeValue** – value (null for elements)
 - someEl.**parentNode**
 - someEl.**childNodes**
 - someEl.**attributes**
 - Methods – actions, like add/delete an HTML element
 - someEl.**getElementById(id)**
 - someEl.**getElementsByTagName(name)**
 - someEl.**appendChild(node)**
 - someEl.**removeChild(node)**
- Advantages
 - XML/Tree structure makes it easy to traverse
 - Elements can be accessed 1+ times
 - XML/Structure of the Tree is modifiable
 - Values/Elements/Structure can be added, changed, and modified
 - A standard of the W3C
- Disadvantages
 - Resource-intensive, consuming lots of memory
 - Needs to be fully loaded in main memory
 - Can be slow
 - Depends on the size and complexity of the tree
 - May not be best choice for all devices/apps
 - Graphics-intensive apps/games
 - A better alternative – using Canvas directly

- e.g., OpenGL

Monitoring User Interaction

- Event Object
 - Each event has an associated object
 - Part of the DOM
 - Event Object provides info about:
 - target element in which the event occurred
 - state of the keyboard keys
 - location of the mouse cursor
 - state of the mouse buttons
- Event Handling
 - Like any interactive app, events can be caught and used to execute functions
 - e.g., user input from forms can be validated on the client-side using JavaScript
 - **onsubmit="return validateForm()"**
- Event Flow
 - Each event object has an 'Event Target', e.g., any node in the DOM tree from where an event originated
 - 2 types:
 - event capture (*global handling*)
 - Event **propagates downwards** through an element's ancestors
 - Any event listeners of the ancestor elements will be executed first
 - even bubbling (*local handling*)
 - Event **propagates upwards** through an element's ancestors
 - Any event listeners of the element will be executed first
 - Ancestors can then potentially handle the event
 - Follows a "Round Trip" pattern/model

Week 8

Lecture 13: Client-Side Scripting (JavaScript)

JavaScript

- Intro/General
 - Not the same as Java
 - Good/bad marketing idea
 - Typecasting
 - Web only
 - Designed to run in Netscape Navigator
 - Became standard in all browsers
 - Useful for a wide range of programming tasks (e.g., node.js)
 - Looks procedural, but closer to functional
 - Functions are first class (can treat functions as variables – pass, return, store, etc)

- Supports anonymous functions (heavily used by jQuery)
 - Moving Target
 - Opinions formed on earlier versions
 - which lacked OO and exception handling
 - There is a standard (ECMA)
 - 11th edition
 - Design Errors
 - Small annoying things (semi-colon insertion, overloaded operators)
 - IDEs can check syntax, or JSLint
 - Object-Oriented
 - Has objects, which encapsulate data/methods
 - No classes, no inheritance
 - Lousy Implementations
 - Engines of early browsers were buggy
 - Browsers containing engines were buggy
 - JS performance war has helped
 - Amateurs
 - Most people writing JS are not programmers
 - Lack of training, discipline, common sense
 - Expressive language that is severely underutilised
 - Conclusion – The Misunderstood Language
- Core features
 - Syntactically similar to Java/C (if/else, while, for)
 - Familiar primitive datatypes (numbers, strings, Booleans)
 - OO (in its own way)
 - Interpreted Language (no compiling)
 - Dynamic Typing (weakly typed: var x = 10; var y = “abc”;;)
 - Functions are first class (can also be anonymous and nested)
 - Used in 3 ways (like CSS):
 - Inline
 - With HTML (<script> tag) or CSS
 - Good for experimentation
 - Violates separation of concerns
 - Embedding
 - Scripts can also be added in HTML head
 - Add JS to event handlers, e.g., to call functions
 - Fragile to maintain
 - External
 - .js
 - Linked to from the <head> section
 - Easier to manage code (over time)
 - DOM Integration
 - Intent behind JS was to dynamically script/manipulate docs
 - HTML docs are modelled using DOM
 - DOM methods and properties can be accessed and altered using JS
 - Finding Elements
 - getElementByTagName()
 - getElementById()


- Modifying Elements

```
// This function traverses the DOM tree and
// converts all Text node data to uppercase
function upcase(n) {
  if (n.nodeType == 3 /*Node.TEXT_NODE*/) {
    n.nodeValue = n.nodeValue.toUpperCase();
  } else {
    // If the node is not Text, loop through its children
    // and recursively call this function on each child.
    var kids = n.childNodes;
    for (var i = 0; i < kids.length; i++) {
      upcase(kids[i]);
    }
  }
}
```

-
- nodeType return type of node
 - 1 – element node
 - 2 – attribute node
 - 3 – text node
 - 8 – comment node
 - 9 – document node
- Include a reference to script containing upcase in HTML head
- Call the function by putting the following at the **bottom** of the doc body:
 - `<script type="text/javascript">`
 `upcase(document.body)`
 `</script>`

Syntax of JS

- Lexical Structure
 - Case-sensitive (keywords, identifiers, variables, functions, etc must be consistent)
 - Whitespace (spaces, tabs, and newlines) is ignored but...
 - Semicolons are optional, but good practice
 - JS interpreters automatically add them (explicit – better)
 - ```
return
true;
```



```
return;
true;
```
  - (returns **undefined**, not **true**)
  - Comments can be single (//) or multiline (/\* \*/)
  - Literals – data values that appear directly in the language (12, 1.2, “hello”, true, false)
  - Identifiers – names for variables and functions
    - First char must be letter/underscore/dollar
    - Remaining chars – above and numbers
  - Reserved word set cannot be used as identifiers (if, return, etc)
    - JS has an unusually large set of reserved words
- Datatypes
  - Primitive:
    - **Number**: no distinction between integers and decimal, floating-point values
    - **String**: sequence of Unicode letters, digits, and punctuation chars delimited by single/double quotes (“Hello!”)
    - **Boolean**: true or false

- Trivial:
  - **null**: an assignment value that can represent no value – null is (a placeholder for) an object
  - **undefined**: variable that has been declared but no value has been assigned to it, or an object property that does not exist
- Functions
  - \* - piece of executable code that is defined once but can be called multiple times
  - In JS, functions are first-class objects and can be passed as datatypes
  - No return type required in function signature

```
function square(x) {
 return x * x;
}

y = square(4);
```

```
var sq = function(x) {
 return x * x;
};

function applyOperator(op, x){
 return op(x)
}

y = applyOperator(sq,4);
```

- Objects
  - \* - collection of named values (**Properties**)
  - Created by invoking a **constructor** or using the **object literal** short-hand syntax

```
function point(xVal, yVal) {
 this.x = xVal;
 this.y = yVal;
}

var p1 = new point(2.5, 5.4)
```

```
var p1 = new Object();
point.x = 2.5;
point.y = 5.4;
```

```
// same as above
var p1= {x:2.5, y:5.4};
```

- Arrays
  - Act as a collection of data values
  - Each value has an index
  - Elements do not have to have the same type
  - Dynamic size
  - Methods: join, reverse, sort, concat, slice, splice, push, pop

```
var collection = new Array();
collection[0] = 120;
collection[1] = 'hello!';

// array literal syntax, same as above
var collection = [120, 'hello!'];
```

- Variables
  - \* - identifier associated with a value
  - Used to store and manipulate values in a program
  - Untyped (weak/loose typing)
  - Declared using **var**
    - If missing, global (not recommended)
    - var i = 10;
    - i = "hello!";
  - Scope depends on where declared
    - Global vars can be seen everywhere

- Variables declared in a function are only visible locally
- Omitting **var** in functions will use matching global vars
- There is **no block scope** (C, Java)

```
var i=10;
var j=10;
function scope() {
 i="hello";
 var j="hello";
}

scope();
// i is "hello"
// j is 10
```

- Expressions

- \* - phrase of code that can be evaluated to produce a value

```
1.5 // a numeric literal
"hello!" // a string literal
True // a boolean literal
/java/ // a regular
// expression literal
{x:1.2, y:2} // an object literal
[1, 2, 3, 4, 5] // an array literal
function(x) {return x*x;} // function literal
sum // the variable sum
```

- Operators

- Combining expressions
- JS supports a common set of ops compared to other C/Java languages
  - arithmetic (+), equality (==), relational (>), logical (&&)
- Care should be taken when using ops
  - + - can mean addition or concatenation
  - == tests for equality, === equality and type
    - true == 1 (true)
    - true === 1 (false)

## Lecture 14: JS+, jQuery

### JS+

- Statements

```
// expression statement
var x = 1 + 2;

// if/else if/else
// statement
if (condition) {
 statements
}
else if (condition) {
 statements
}
else {
 statements
}

// while statement
while (condition) {
 statements
}

// for statement
for (init ; test ; inc) {
 statements
}

// function statement
function name (args) {
 statements
}
```

- }
- Try-catch:

```

 // try catch finally statement
 try {
 // normally this code runs from top to bottom
 // sometimes an exception may be thrown
 // either directly with a throw statement,
 // or indirectly by calling another method
 }
 catch (e) {
 // the statements here are executed if, and only
 // if, the try statement generated an exception
 // these statements handle the exception somehow
 }
 finally {
 // the statements here are always executed
 // regardless of what happened in the try block
 }
}

```

- Addressing objects
  - Unordered collections of properties
  - ‘.’ can be used, but also ‘[]’
    - Similar to dictionaries in Python or Hashtable in Java
  - As JS is dynamic and objects can have properties added at any time, method is very convenient

```

var cust=new Object();
cust.addr0="36 King St";
cust.addr1="42 Queen Rd";
cust.addr2="16 Abbey St";

```

```

var addr = "";
for (i = 0; i < 3; i++) {
 addr += cust["addr" + i] + '\n';
}

```

- 
- Functions+
  - Can be nested
  - Support optional args
    - if invoked with fewer args, undefined is used
  - The args object can be used with variable length argument lists
    - e.g., function object has a property **arguments**, which can be inspected to find which and how many arguments were given (if(arguments.length != 3){})
  - Functions that are properties of objects are usually referred to as **methods**
- Classes, Constructors
  - JS – protocol-based OO
  - Creating a class to model Rectangles:

```


class Rectangle {
 // Define the constructor
 // Note how it calls a method referred to by "this"
 constructor (idString, widthVal, heightVal) {
 this.id = idString;
 this.resize(widthVal,heightVal);
 }

 // What follows is a method
 resize (widthVal, heightVal) {
 this.width = widthVal;
 this.height = heightVal;
 }

 // Here is another method
 getArea () {
 return this.width * this.height;
 }
}

// Test out the constructor and methods
var rect = new Rectangle ("Test", 4, 5);
document.writeln(rect.id);
document.writeln(rect.getArea());
rect.resize(6, 7);
document.writeln(rect.getArea());

```



- - Regular Expressions
    - \* - an object that describes a pattern of chars that can be used for **pattern matching** and **search and replace** actions
    - Often thought of as programs within a program
    - Despite utility, documentation is bad
    - In JS, represented by RegExp objects
      - Syntax:
        - /pattern/modifiers;
        - var re1 = Free/i;
          - i – case insensitive
        - var re2 = /s\$/;
          - match any string that ends with 's'
      - Methods:
        - search() – returns starting position of the first match, or -1
        - exec() – returns first match or null
        - test() – returns true if there is a match, false otherwise
- Problem
  - Despite JS and DOM being functionally useful, coding on the client-side is not easy (consider Java and its huge standard library of useful functionality)
  - DOM scripting entails a lot of repetitive domain-specific boilerplate coding
- Solution:
  - JS needs its own standard library
  - Focus on the domain-specific programming tasks
  - More than 1 solution: **jQuery**, YUI, MooTools, etc

## jQuery

- JS framework that works the same way as JS, but is shorter – “write less, do more”
- Created by John Resig
- One of the most popular JS libraries

- Simplifies client-side scripting:
  - Selecting DOM elements
  - Creating UI animations and effects
  - Handling events
  - Developing AJAX apps
- Cross-Browser Compatibility
  - jQuery takes a lot of the problems out of developing for multiple browsers
  - Acts as a layer of abstraction over various browsers
  - No more browser sniffing
- Plug-In architecture
  - jQuery creates a useful foundation for additional functionality to be added
  - A wide range of specialised plug-ins have been developed since the release of jQuery for many web-dev tasks (e.g., jQueryUI)
- Syntax `$()`
  - **Selecting** and **acting** on a particular DOM element and manipulating its parameters
  - Selectors of CSS are reused in jQuery

|                          |                    |                                |
|--------------------------|--------------------|--------------------------------|
| <code>\$('#name')</code> | <code>.text</code> | <code>('the new text');</code> |
|--------------------------|--------------------|--------------------------------|

|                      |                   |                                 |
|----------------------|-------------------|---------------------------------|
| <code>\$('p')</code> | <code>.css</code> | <code>('color', 'blue');</code> |
|----------------------|-------------------|---------------------------------|

- |   |        |        |            |
|---|--------|--------|------------|
| ○ | Select | Action | Parameters |
|---|--------|--------|------------|

- Clean, Consistent Markup
  - jQuery reuses the pattern
  - Heavy use of anonymous functions
  - Chaining functions together possible
- Page 'readiness'
  - **`(document).ready`** ensures code inside isn't executed until **after the page has loaded**

```
$(document).ready(function() {
 alert('Hello World!');
});
```

- 
- Shorthand: `$()`

```
$(function() {
 alert('Shorter form!');
});
```

- 
- Events

- Syntax:

|                      |                     |                                  |
|----------------------|---------------------|----------------------------------|
| <code>\$('p')</code> | <code>.click</code> | <code>(function() {...});</code> |
|----------------------|---------------------|----------------------------------|

- |   |        |       |        |
|---|--------|-------|--------|
| ▪ | Select | Event | Action |
|---|--------|-------|--------|

- Types:

- Mouse: click, dblclick, mouseenter, mouseleave, hover
- Keyboard: keypress, keydown, keyup

- Form: submit, change, focus, blur
- Doc Window: load, resize, scroll, unload
- [examples in slides]
- Multiple Events: on()

`$( 'p' )`

Select

`.on`

on

`( { click :  
function() {...}, ... } );`

Event: Action

- Useful plugins
  - validate() for forms
  - Demos in slides: jQueryUI, interactions, widgets, effects, Fundamentals

## Week 9

### Lecture 15: XML & JSON

#### Markup Languages:

- SGML
  - HTML
  - XML
    - XHTML

#### XML

- eXtensible Markup Language
- Developed by the W3C; 1.0 in 1998
  - W3C – a consortium with hundreds of members including the major vendors and users of the web:
    - AT&T, BBC, Citibank, Microsoft, Oracle Xerox
    - quite a few universities
    - founded and led by Sir Tim Berners-Lee
- Designed to **transport** and **store** data
- Design goals emphasise simplicity, generality, and usability
  - Why design XML?
    - Markup for the web was not being properly supported
      - Standard Generalized Markup Language (SGML) was too complex
      - While HTML was too limited and mixed format with structure
  - XML aimed to:
    - Provide a **simpler markup language** (easier than SGML)
    - **Separate format from structure** (Separate Concerns)
    - Be **extensible** and provide support for a host of apps
    - **Transport and store data**
- Role
  - To describe the structure of **semi-structured docs**
  - A mechanism **for sharing, transporting, and storing annotated data**
  - To be a general-purpose language for **data description** and **interchange**

- i.e., forms the basis of other languages
  - XML has:
    - emerged as a dominant standard
    - developed a number of **vocabularies** for specific disciplines
    - additional tools for additional layers of processing, such as:
      - the **separate** ability to add **formatting** to XML docs
      - querying XML docs, transforming XML docs, etc
- Extensions:
  - XHTML – web pages
  - Wireless Markup Language (WML) – a specialisation of XML for Wireless Application Protocol – for early mobile data
  - MathML – language of Maths
  - Chemical Markup Language – “HTML with Molecules”
  - SOAP – for describing distributed method parameters
  - etc
- Structure
  - General:
    - HTML was designed to *display* data
      - HTML elements mix format and structure with content and presentation
      - In XML, **tags define structure**, and any presentation is handled separately
    - **Structure** of XML is **tightly controlled**:
      - Tags are **case-sensitive**, and variable values must be **quoted**
      - If there is a **start tag**, there must be an **end tag**
      - A hierarchical structure of elements is enforced
      - (not strictly enforced in HTML)
    - XML provides flexibility
    - **New tags** (and variables) can be created, i.e., user-defined
  - Document:
    - Optional **prologue – XML declaration**
      - `<?xml version="1.0" encoding="UTF-8"?>`
      - Version – must be 1.0 or 1.1
      - Encoding – how characters are encoded in the file
      - Standalone – “yes” if the doc is entirely self-contained, “no” if it has an external DTD/Schema (default – “no”)
    - **Body** – contains the doc elements and data
    - Optional **epilogue** – contains **comments** and **processing instructions**
      - `<!--This XML document is over -->`
- Elements
  - Basic building blocks of XML
  - Is everything from (including) the element’s start tag to (including) the element’s end tag
  - Can contain text, attributes, other elements, or a mix of the above
  - Names are case-sensitive
  - Closed elements consist of both opening and closing tags:
    - `<Url>http://www.gla.ac.uk/</Url>`



- Can be nested
    - All elements must be nested within a single root element
    - Nested elements are child elements
  - Empty elements are denoted by `<Url></Url>` or just `<Url />`
- Attributes and Values
  - Attributes – characteristics of elements
    - Case-sensitive
    - Have values – must be in quotes
  - All values are text strings
    - Values can contain most characters and whitespace
      - Take care when using special characters, esp. `<`, `>`, `"`, etc – use escape values, e.g., for `<` use `&lt;`
- Well-Formed XML if:
  - **XML tags are Case-Sensitive**
  - **Corresponding tags:** for every start tag there is an end tag
  - **Hierarchically structured:** an XML parser will be able to process it and make use of the tree structure
    - e.g., `<a><b>some text</a></b>` - not well-formed, i.e., not properly nested
  - XML **attribute values** must be **quoted**
  - XML **docs** must have a **root** element
- Predefined and Valid
  - To share XML, a **pre-defined structure** can be used:
    - These describe the tags which can appear, and can be done using:
      - **1. Document Type Definitions (DTD), or**
      - **2. XML Schemas and XML Namespaces**
    - The XML can be checked according to the definitions and validated
    - These structures are references either at the top of the file or provided separately
  - An XML doc is **valid** if it is well-formed and conforms to the rules in the DTD/Schema
  - Many XML validators available, e.g., [www.xmlvalidation.com](http://www.xmlvalidation.com)
- DTDs vs Schemas (examples in slides)
  - Schemas are more powerful than DTDs:
    - Written in XML
    - Extensible to addition
    - Support data types
    - Support namespaces
  - Why use XML schema?
    - XML files can carry a description of its own format
    - Independent groups of people can agree on a standard for interchanging data
    - Ability to verify data
- Formatting (non-examined)
  - XSL (Extensible Stylesheet Language) – styling language for XML
  - XSLT (XSL Transformations)
    - Can be used to transform XML docs into other formats (e.g., into XHTML)
  - Process
    - Start with a raw XML doc

- Create an XSL Style Sheet
- Link it to the XML doc
  - e.g., `<?xml-stylesheet type="text.xsl" href="menus.xsl"?>`

## XHTML

- Strict
  - Descends from XML, so rules to follow
    - Separates visual rendering from the content
      - No style tags
    - Strict set of rules enforced on markup
      - e.g., Hierarchy enforced strictly, tags all lowercase, restricted placement of elements
    - An XHTML Strict doc will work in many different environments
      - Visual browsers, braille readers, text-based browsers, print
    - Highly configurable by the user
    - Highly maintainable by dev

## Differences between XML and HTML

- XML was designed to **transport** and **store** data
- HTML was designed to **display** data
- => **Carrying** info vs **displaying** info

## JSON (JavaScript Object Notation)

- In general
  - Lightweight data interchange format
    - Easy for humans to read, write
    - Easy for machines to parse, generate
    - Less boilerplate => more info per byte
  - Built on 2 universal data structures
    - A collection of name/value pairs
      - Often realised as an object/record/struct/dictionary/hash
    - An ordered list of values,
      - Often realised as an array/vector/list
  - Language-independent
- Compared to JS
  - JSON uses JS syntax, but the JSON format is text-only (like XML)
  - Text can be read and used as a data format by any programming language
  - Evaluates to JS Objects
    - JSON format is syntactically identical to the code for creating JS objects
    - Instead of using a parser (like XML does), a JS program can use standard functions to convert JSON data into native objects
- Syntax
  - Derived from JS object notation syntax:
    - Data is in name/value pairs in the form "name": "value"
    - Data is separated by commas
    - Curly braces hold objects
    - Square brackets hold arrays

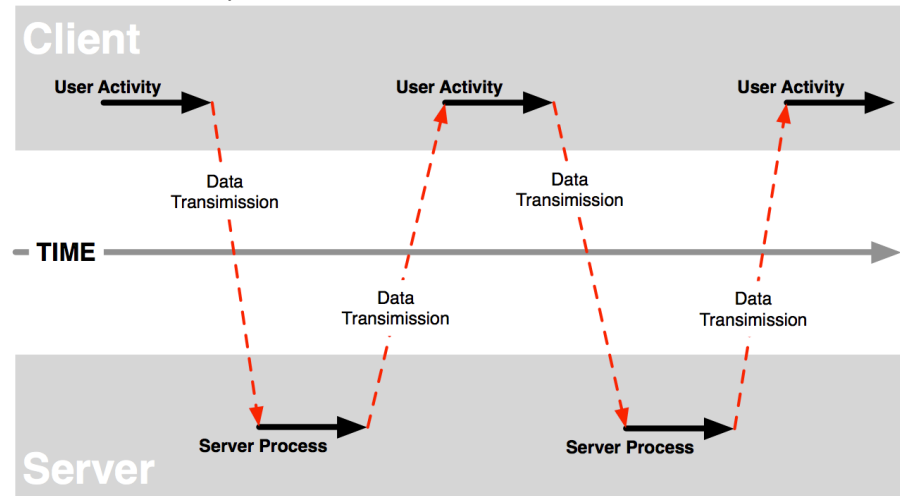
- Example JSON object
  - {"firstName": "John", "lastName": "Doe"}
- Example JSON array:
  - [{"firstName": "John", "lastName": "Doe"}, {"firstName": "Anna", "lastName": "Smith"}]
- [more examples in slides]
- Possible to implement in Python
- Versus XML
  - Similar:
    - "Self-describing" (human readable)
    - Hierarchical (values within values)
    - Can be parsed and used by lots of programming languages
    - Can be fetched with an XMLHttpRequest (see AJAX)
  - Different:
    - JSON doesn't use end tags
    - JSON is shorter
    - JSON is quicker to read, write
    - JSON can use arrays

## Lecture 16: AJAX

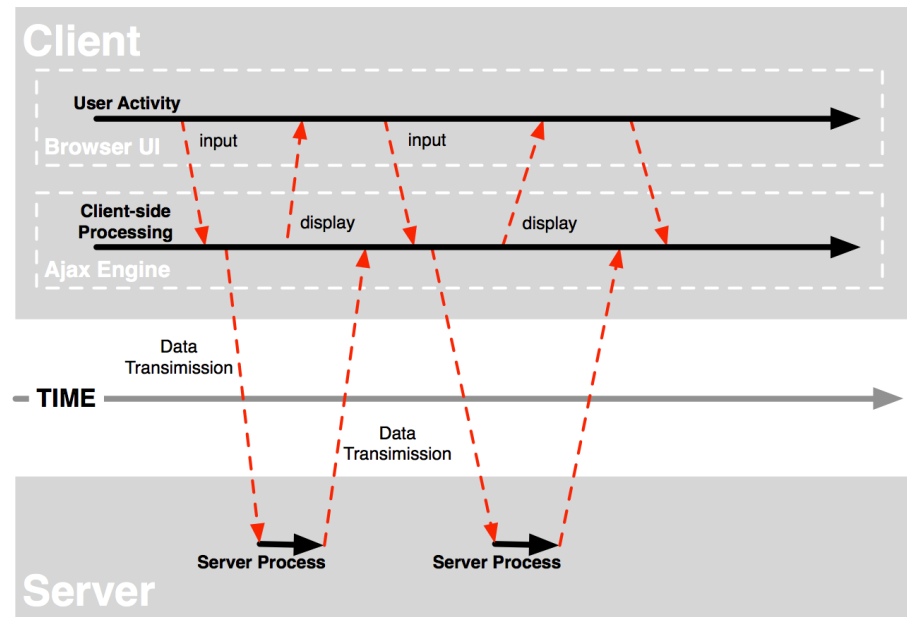
### AJAX (Asynchronous JS And XML)

- Origins
  - A key technology set underlying web apps
  - Jesse James Garrett coined the term in "Ajax: A New Approach to Web Applications" (2005)
  - Critically, all of the components have existed in some form since the late 1990s
    - An example of where browsers introducing non-standard features has been a positive thing
  - Generated an enormous amount of hype and energy in web dev ever since
- In general
  - AJAX **eliminates the need to reload** a web app in order to get new content from the server
  - This removes the **start-stop interaction** where a user has to wait for new pages to load
  - An intermediate engine (**AJAX Engine**) is introduced into the communication chain between client and server
  - Improves the interactive experience in web apps
- Technology Set
  - Garret's OG technology set:
    - Standards-based presentation using (X)HTML and CSS
    - Dynamic display and interaction using the Document Object Model
    - Data interchange and manipulation using XML and XSLT
    - Asynchronous data retrieval using XMLHttpRequest
    - JavaScript binding everything together

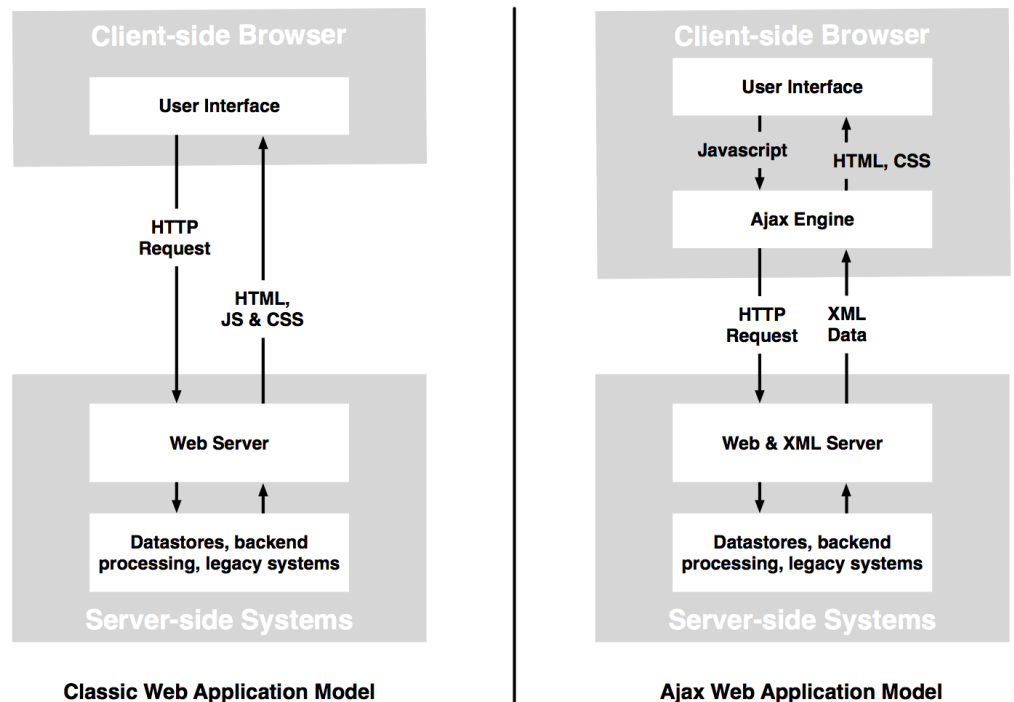
- Can write in 'pure' JS, jQuery, or newer *fetch* API
- The Mechanics
  - Traditional Client/Server Synchronous Communication Model



- 
- AJAX Components
  - JS can manipulate the DOM of a webpage to **create, modify and remove content and style**
  - JS event handlers can be attached to events generated **by the user and browser**
  - XML can **model data** and it can be accessed using the DOM (AJAX can also transport JSON or plain text)
  - XMLHttpRequest Object
    - Keystone of AJAX
    - Introduced by Microsoft in IE 5
    - An object that is part of the DOM and is built into most modern browsers
    - Can communicate with the server by sending HTTP requests (much like normal client/server communication)
    - Independent of <form> or <a> elements for generating HTTP GET/POST requests
    - Does not block script execution after sending an HTTP request
    - As with **content** and **style**, JS can now programmatically manage **HTTP communication**
- AJAX Client/Server Asynchronous Communication Model



- Architectural Comparison:



- How it Works:

- 1. An event occurs in a web page (the page is loaded/a button is clicked)
- 2. An XMLHttpRequest object is created by JS
- 3. The XHR object sends a request to a web server
- 4. The server processes the request
- 5. The server sends a response back to the web page
- 6. The response is read by JS
- 7. Proper action (like page update) is performed by JS

- XMLHttpRequest properties:

- readyState**
  - Holds the status of the XHR
    - 0: request not initialised

- 1: server connection established
  - 2: request received
  - 3: processing request
  - 4: request finished and response is ready
- **onreadystatechange**
  - accepts an EventListener value, specifying the method that the object will invoke whenever the readyState value changes
- **status**
  - Represents the HTTP status code and is of type short (e.g., 200 = OK, 404 = Not Found)
- **responseXML**
  - Represents the XML response data when the complete HTTP response has been received (when readyState is 4), and when the Content-Type header specifies the MIME (media) type as text/xml, application/xml, or ends in +xml
- **responseText**
  - contains the text of the HTTP response received by the client
  - XML is not the only method to model data in AJAX apps. A popular alternative is JSON
- XHR methods:
  - **open(method, url, async, user, psw)**
    - specifies the request
      - method: GET/POST
      - url: file location
      - async: true (asynchronous) or false (synchronous)
      - user: optional username
      - psw: optional password
  - **send()**
    - sends the request to the server
    - used for GET requests
  - **send(string)**
    - sends the request to the server
    - used for POST requests
  - **abort()**
    - cancels the current request
- [examples in slides]
- Sending a Request
  - GET or POST?
    - GET is simpler and faster than POST, and can be used in most cases
    - Always use POST requests when:
      - a cached file is not an option (update a file/database on the server)
      - sending a large amount of data to the server (POST has not size limitations)
      - sending user input (which can contain unknown chars), POST is more robust and secure than GET
    - Example of a GET request:



## Week 10

### Lecture 17: Processing XML, DOM/SAX Parsing

#### XML Structure (recap)

- XML has a tightly controlled structure; documents must follow a number of rules to be *well-formed*
- Case sensitive `<start_end>tags</start_end>`
- Obeys a **hierarchical structure**
- All docs have a **root** element
- Might also be DTD/Schema rules to validate against

#### Displaying XML

- Valid XML can be displayed in a browser
- Browser might give an error in the case of invalid XML

2 main ways to use XML in a program:

- **DOM** (Document Object Model)
  - builds an **in-memory hierarchical model** of the XML elements
  - appropriate if you need the **whole doc** or need to **move about it freely**
- **SAX** (Simple API for XML)
  - provides an **event-driven parser** for XML
  - appropriate for using **parts of the data** in the order they appear in the file, or if there are memory constraints

#### DOM

- In general
  - W3C standardised for accessing docs
  - “The W3C **Document Object Model** is a **platform** and **language-neutral interface** that allows programs and scripts to **dynamically access** and **update** the **content**, **structure**, and **style** of a **document**.”
  - Separated in 3 main parts:
    - Core DOM: standard model for any structured doc
    - HTML DOM: standard model for HTML docs
    - XML DOM: standard model for XML docs
- XML DOM
  - Standard model and programming interface for XML
  - Defines **objects** and **properties** of all XML elements along with the **methods** to access them
    - **The standard for getting, changing, adding, and deleting XML elements**
  - **DOM defines everything in an XML doc as a node**
  - Nodes:
    - XML doc is a document node
    - Every XML element within the doc is an element node



- The root element of the doc is the root node
  - Even the text of XML elements is a node (text node)
- An advantage of a tree structure is that it can be traversed without knowing the exact structure and without knowing the type of data it houses
- Working:
  - In whichever language/environment one works, the technique is the same:
    1. load the XML doc object
    2. locate the root element or some other element that is of interest
      - either traverse the tree
      - or search for the desired element
    3. for the given element:
      - extract the attributes and their values
      - extract the element data
      - and/or add/modify/remove elements or attributes
    4. go to step 2 and repeat
- Parsing:
  - Creation of DOMParser object
  - Parsing of XML into xmlDoc object
  - Traversal of DOM node tree
  - In AJAX examples, no need to parse first; obtained DOM via xhttp.responseXML
  - [lots of examples in slides, including parsing with AJAX and Java]

## SAX Parsing

- SAX (Simple API for XML)
  - Sequential access parser API for XML
  - Not an alternative to DOM
    - No default object model
    - Another mechanism for reading XML
  - **Stream parser**, which is **event-driven**
    - **Parsing is unidirectional, i.e., no going back**
    - **Callback methods** are **triggered** by **events** when parsing
  - Oriented towards **state-independent processing**
    - An alternative to SAX is **StAX**, which is oriented towards state-dependent processing
- Event handling
  - Events are available for XML...:
    - text nodes
    - element nodes
    - comments
    - processing instructions (often used in XPath and XQuery)
  - Events are triggered when:
    - open/close element tags are encountered
    - data (#PCDATA and CDATA) sections are encountered
    - processing instructions, comments, etc are encountered
- Working:
  1. **Creating a custom object model**

- like ResultSet and Result
- 2. **Creating a SAX parser**
- 3. **Creating a DocumentHandler** to turn the XML doc into instances of the custom object model
  - ContentHandler: implements the main SAX interface for handling doc events
  - DTDHandler: handles DTD events
  - EntityResolver: resolves external entities
  - ErrorHandler: reports errors and warnings
  - DefaultHandler: for everything else
- Handler Methods
  - *startDocument*
    - performs any work required before parsing
  - *endDocument*
    - performs any work required at the end of the parsing (like reporting analytical results)
  - *startElement*(name, attributes)
    - perform any work required when the start tag of an element of that name is encountered
  - *endElement*(name)
    - perform any work required when the end tag of an element of that name is encountered
  - *characters*(ch)
    - perform any work required when a text node is encountered

#### DOM vs SAX

- DOM
  - Uses more memory
  - Tends to be slower
  - Can handle parsing that requires access to the entire doc
    - (if it fits in memory)
  - Easier to program
  - Can process files larger than main memory through disk caching
    - but this is extremely slow
- SAX
  - Uses less memory
  - Tends to be faster
  - Can process files that are larger than main memory
  - Requires more programmer effort
  - Cannot handle all parsing tasks directly, i.e., if all XML is required for validation
    - would need multiple parses

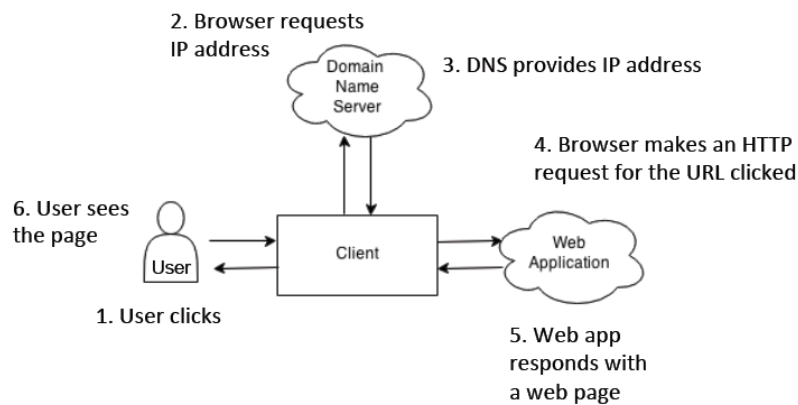
## Lecture 18: Messaging

### At the App Layer

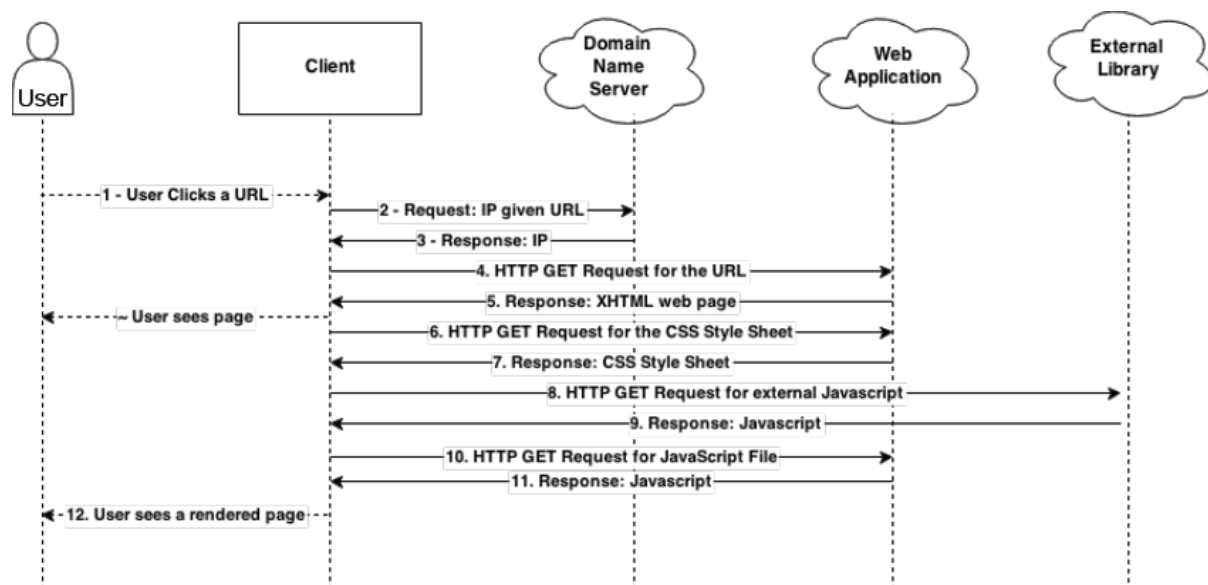
- **Hypertext Transfer Protocol (HTTP)**

- App protocol used to send messages
- Specific URL scheme
- Follows a **Request-Response** (Request-Reply) pattern
  - Way to exchange messages
  - A requester sends a request message and the receiver of that message provides a message in response
  - Typically, this is performed in a synchronous fashion (as in HTTP)
    - Can be asynchronous (e.g., HTTP/2)
  - Process:
    - When the **user agent** (i.e., the web browser/client) is asked to **send a message** (i.e., when the user clicks a link):
    - First, the URL is turned into an **IP address**
      - **Request:** ask Domain Name System for IP
      - **Response:** returns the IP for the URL
        - e.g., [www.gla.ac.uk](http://www.gla.ac.uk) maps to 130.209.34.12
    - Second, a **TCP connection** is opened on a particular port on the node at that IP address
      - port 80 for HTTP (standard)
      - port 443 for HTTPS (encrypted through TLS)
    - Next, a request is made using a specific URL Scheme (e.g., HTTP) and sent using that TCP connection
      - **Request:** Get the homepage of the specified URL
      - **Response:** Returns the XHTML for the home page
      -
    - Provides a number of ways to make a request (GET, POST)
- User Agent Specific Protocols (
  - Package/wrap the info that will be sent when providing a response
  - Such as XML, XHTML, JSON

## Flow of Messages:



## Sequence Diagrams



- System Architecture Diagram aggregates over all messages, only showing the flow, but hides a lot of detail
- Sequence Diagrams provide a better way to show the flow of messages
- Label all messages – the more precise, the better

## Communicating with the Database

- Web Apps typically make requests to a Database
- In Django, requests are made indirectly, through the Object Relational Mapping (ORM)
  - The actual request may be via HTTP or some other protocol
  - But it can be specified as an ORM Request and ORM Response

## Protocols

- Requests can be made using various protocols:
  - **http**: common, indicates a file that a web browser can format and display – HTML file, image file, sound file, etc

- **https**: utilises Transport Layer Security (TLS) for secure communication and always sends data in encrypted form
- **file**: indicates a file which is not in a recognised web format and will be displayed as text
- **ftp** (file transfer protocol): used to refer to websites from which files can be extracted and downloaded to the client machines
- **mailto**: if selected, such a link generates a form in which an email message can be constructed and sent to a designated user
- **news**: the resource is a news group or article
- **telnet**: generates a telnet session to this server
- **HTTP (Hyper Text Transfer Protocol)**
  - Used to deliver virtually all files and other data using **8-bit characters**
    - Usually, HTTP takes place through TCP/IP sockets
  - Used to transmit **resources**, not just files
    - Resource – some chunk of info that can be identified by a URI
  - Functions as a **request-response pattern** in the **client-server** computing model
    1. An HTTP client opens a connection and sends a **request message** to an HTTP server
      - Typically, the request is GET/POST
    2. The server then returns a **response message**, usually containing the resource that was requested
      - Typically, in XHTML, XML, but also in other formats like JSON
    3. After delivering the response, the server **closes the connection**, making HTTP a stateless protocol
      - i.e., not maintaining any connection info between transactions
  - GET
    - Appends the data to the URL as key-value pairs
      - URL ? key1 = value1 & key2 = value2
    - Special characters within the values are replaced, e.g., %20 for space
      - **URL-encoding**
    - The user can see, copy, and bookmark a URL; thus, it is easy for them to 'resubmit' the page
    - Therefore, GET should be used for pages which don't change anything on the server
      - e.g., fine for info requests
        - GET  
pro.pl?name=John%20Smith&address=5%20Queen%20Street HTTP/1.1
        - GET /base/query/?query=big+java HTTP/1.1
  - POST
    - **Sends data packaged as part of the message**
      - must be used for multipart/form-data, e.g., file uploading
      - should be used for programs with side effect
        - e.g., database update, purchase requested, sending email
      - or if there are non-ASCII chars in the data (e.g., accented letters)
      - if the data set is large (GET can have problems with >1kb)
      - or hide data from users (although they can always view the source)

- Uses the message body to achieve this and has the following header lines:
    - Content-Type: application/x-www-form-urlencoded
    - Content-Length: 26 // number of characters
    - body, e.g.,
      - name=John%20Smith&address=5%20Queen%20Street
      - (referred to by x-www-form-urlencoded)
- GET vs POST
  - Use **GET** for **safe** and **idempotent** requests
  - Use **POST** for requests that are **not** safe or idempotent
  - **Safe** operation – does not change the data requested
  - **Idempotent** operation – the result will be the same no matter how many times it is requested
- Other methods:
  - HEAD – like GET, except it asks the server to return the response headers only
    - Useful to check characteristics of a resource without actually downloading it
  - PUT – for storing data on the server
  - DELETE – for deleting a resource on the server
  - OPTIONS for finding out what the server can do, e.g., switch to secure connections
  - TRACE for debugging connections
  - CONNECT for establishing a link through a proxy
- Stateless Communication
  - HTTP doesn't require the server to retain any info about the client/user
    - All requests are independent
  - **Common Solutions** to overcome Statelessness:
    - **Client-Side:** use HTTP Cookies
      - Cookies – tokens stored on the client and can be included in the request
      - Best to store a session ID in the cookie that the server can use to retrieve info about the user (rather than actual data about the user, etc on their client)
    - **Server-Side:** with hidden variables when the page is a form
      - i.e., through POSTs
    - **URL encoding:** store a session ID within the URL

## Week 11

### Lecture 19: MVC, Web App Frameworks

#### Re-inventing the Web App

- After developing several web apps (from scratch), it rapidly becomes clear that:
  - there is lots of coding overhead and 'boiler plate' code
  - typically, the same tasks are repeated

- e.g., access a database, process then present results in HTML
- there is a need for separation of concerns
  - distribution of the main components/interactions to maximise code reuse, provide robustness, aid in debugging, enable scalability, etc

### Prefabricated Wheels

- Web frameworks typically provide (some of) the following features:
  - User authentication, authorisation, security
  - Database abstraction (or Object-Relational Mapping)
  - Template system
  - AJAX sub-framework
  - Session Management
  - An Architecture usually based on **Model-View-Controller**

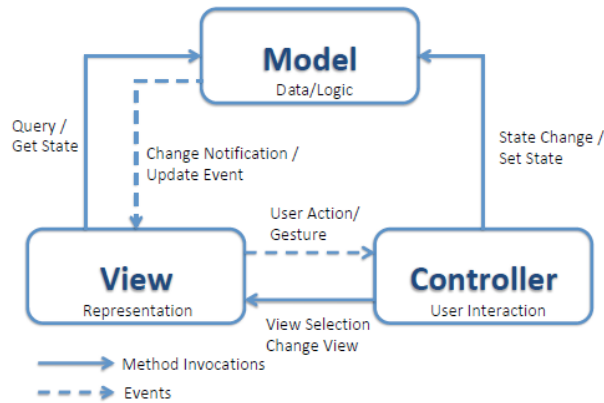
### Design Pattern

- Serves as a tool to **communicate** ideas, solutions, and knowledge about **commonly recurring problems**
- User interface design patterns help designers and developers create the most effective and usable interface for a particular situation
- Thus, each pattern is a 3-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution**
- Patterns can be expressed hierarchically, with each layer representing a different level of granularity, and there may be different ways to (physically) implement each pattern

### Model View Controller pattern

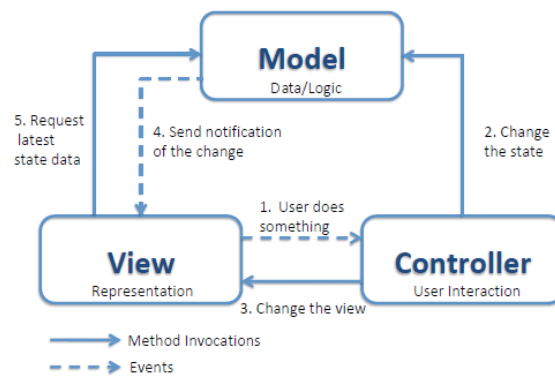
- Separates concerns between:
  - presentation logic
  - business/app logic
  - data model
- Maps the traditional input, processing, output roles into the Graphical User Interface realm (GUI)
- Invented in the 1970s at Xerox Parc (where GUI was developed)
- The **controller** interprets m+kb inputs and maps these to actions
- These commands are sent to the **model/view** to enact the appropriate change
- The **model** manages the data elements
  - responding to queries about its state,
  - and updating its state
- The **view** manages the display for presenting the data
- **Model** (app layer)
  - Any actions wanted to be executed on the raw data must go through this layer. Definitions of how the app works with data (commonly CRUD: create, read, update, or delete) are written here
- **View** (presentation layer)
  - Defined how the pages should look to the user, how the app presents data, or how a user can submit certain instructions to be executed by the app

- **Controller** (orchestrator of the app)
  - Controls the flow of the program. Receives user commands, processes them, and then contacts the model, and finally instructs the view to display appropriately to the user
- Architecture



- Model
  - Represents app data and domain logic
  - Notifies views when it changes and enables the view to query the model
  - Allows the controller to access app data functionality encapsulated by the model
- View
  - Visual representation of its model, presentation filter
    - renders the contents of the model
    - Specifies how the model data should be presented
  - Attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions
    - When the model changes, the view must update its presentation
      - Push Model: the view registers itself with the model for change notifications
      - Pull Model: the view is responsible for calling the model when it needs to retrieve the most current data
  - Responsible for forwarding user requests/gestures to the controller
- Controller
  - Defines the app behaviour
  - Link between a user and the system
  - Interprets user requests/gestures and maps them into actions
    - for the model to perform and
    - arranges for relevant views to present themselves in appropriate places on the screen
- Sequence of actions:





- Advantages:
  - Enables **independent** development & testing
  - Easier to **maintain**
  - Provides **reusable** views & models
  - **Synchronised** views and multiple simultaneous views
  - Helps enforce logical **separation of concerns**
- Disadvantages
  - Some initial **overheads** splitting up concerns
    - Increased overheads in development (i.e., 3 classes vs 1)
    - Especially for very **simple** apps
  - Debugging can sometimes be a problem
    - Harder to track down where problem appears
  - **Requires** the developers to **understand** patterns
- In Django
  - Django has MVCT/**MTV**:
    - **Models** describe database
    - **Controller** handled by:
      - Django Framework
      - URL parser, which maps URLs to views, where processing may occur
    - **Templates** describe how the data is presented

## Frameworks

- Software frameworks provide design and partial implementation for a particular domain of apps
- Allow developers to create apps more efficiently by providing **default functionality**, whilst allowing them to **extend and override** to suit their specific purposes
- Definitions:
  - Set of classes that embodies an **abstract design** for solutions to a **family of problems**
  - Set of **prefabricated software building blocks** that programmers can **use, extend, or customise** for specific computing solutions
  - Large **abstract apps** in a **particular domain** that can be **tailored** for individual apps
  - **Reusable software architecture** comprising both **design and code**
- Reasons to use:
  - Virtually all web apps have a common set of basic requirements
    - (user management, security, password recovery, sessions management, database management, etc)

- Frameworks encapsulate thousands of hours of experience, knowledge, and know-how
  - Improved over each iteration, debugged, secured, etc
- Often can handle reasonably high loads and traffic out of the box
- Characteristics
  - **Inversion of Control**
    - Responsible for the app control flow
  - **Default Behaviour**
    - Must provide some 'useful' functionality related to the app domain
  - **Extensibility**
    - Hot spots designed to be extended
    - Allows developer to customise their app specifically for a particular purpose
  - **Non-modifiable Framework Code**
    - Key components cannot be altered
    - Not strictly non-modifiable, but typically just used, though contributions back to the framework are often subject to the framework creators or open-source community
- Advantages:
  - Enables rapid development
  - Concentrate on unique app logic
  - Reduces boiler plate code
  - Already built and tested => increased reliability
  - Increased security (generally)
  - High level of support for basic common functionality
- Disadvantages:
  - Imposes a certain model of development (80% easy, 20% hard)
  - Can introduce code bloat
  - Levels of abstraction generally introduce performance penalties
  - Difficult to overcome the steep learning curve
  - Can be poorly documented
  - A bug/security risk can seriously compromise the app
- Vs Libraries
  - Frameworks are about reusing behaviours by controlling how abstract classes and components interact with each other
    - A framework calls the app code
  - A library is a collection of classes which provide reusable functionality
    - The app code calls the library
- Web App Frameworks
  - Examples:
    - JavaScript
      - Angular, Backbone, Ember, React, Vue, etc
    - Java
      - Spring, Struts, Grails, Google Web Toolkit, etc
    - PHP
      - Symfony, Cake, CodeIgniter, Laravel, etc
    - Python
      - Django, FastAPI, Bottle, Flask, TurboGears, Pyramid, Zope, etc

- Ruby
  - Rails, Camping, Merb, Sinatra, Padrino, etc
- ASP.NET, ColdFusion, C++, Tcl, Ocaml, Scala, Groovy, etc
- Common Functionality:
  - **Web Template System**
    - to provide predefined pages that load dynamic content
  - **Caching**
    - to reduce perceived lag
  - **Security**
    - to provide authentication and authorisation
  - **Database access and mapping**
    - to speed up working with databases and avoid using SQL
  - **URL Mapping**
    - to enable handling of URLs and friendlier URLs
  - **AJAX handlers and handling**
    - to create more dynamic pages that are more responsive
  - **Automatic configuration**
    - to decrease setup hassles, usually uses introspection and/or following conventions
  - **Form Management**
    - to speed up the creation of forms and their handling
- Reasons to use:
  - To reduce 'boiler plate' code in web apps
    - Particularly, access and manipulation of DB (often called CRUD operations) and session management across multiple pages
  - Web apps have matured to a point where software engineering practices (including design patterns and frameworks) are becoming:
    - increasingly useful
    - necessary
    - the norm
- Caveats/Limitations
  - Require an investment in learning the framework
    - **Learning vs Building Trade-off**
  - Sacrifice some flexibility for rapid development
    - **Flexibility vs Efficiency Trade-off**
  - Like client-side libraries, knowledge of one framework does not necessarily transfer to another
  - Early stages of web framework ecosystems
    - There are many competing options currently
    - Eventually, the most popular (few) will emerge