A mix of technologies today: CSS, client- and server-side JavaScript

TI1506: Web and Database Technology
Claudia Hauff

Lecture 6 [Web], 2014/15

Course overview [Web]

1. http: the language of Web communication
2. Web (app) design & HTML5
3. JavaScript: interactions in the browser
4. **node.js: JavaScript on the server**
5. **CSS: Lets make things pretty**
6. **Ajax: asynchronous JavaScript**
7. Personalization: Cookies & sessions
8. Securing your application

Learning objectives

- **Implement** CSS media queries
- **Implement** simple CSS animations
- **Implement** “plain” Ajax (without the help of jQuery)
- **Understand** the connection between Express and Connect
- **Employ** the template language ejs

CSS media queries

Not just one device but many…

- Different devices should be served different styles, e.g.
  - **Printing** a todo list: ideally only b/w, no color blocks
  - **Viewing** a todo list on a small screen: remove non-essential information (footer, etc.)
  - **Viewing** a todo list on a large screen: present all available information
  - **Text-to-speech** devices: remove non-essential information

- CSS media queries enable the use of device-dependent (i.e. media-type dependent) stylesheets

HTML: write once

CSS: write once per device

Media queries can be complex

Media types: **`all`**, **`print`**, **`screen`**, **`speech`**

```
<link rel="stylesheet"
   media="screen and (min-width: 800px),
             (min-width: 3000px)"
   href="large-device.css">
…
<style>
     @media print {
       body {
         color: black !important;
         width: 100%;
       }
     }
     @media screen and (max-width: 300px) {
       #sidebar {
         display: none;
       }
     }
</style>
```

Media queries can be complex

Media types: **`all`**, **`print`**, **`screen`**, **`speech`**

```
1  <link rel="stylesheet"
2    media="screen and (min-width: 800px),
3              (min-width: 3000px)"
4    href="large-device.css">
5  …
6  <style>
7       @media print {
8         body {
9           color: black !importa
10          width: 100%;
11        }
12      }
13      @media screen and (max-width: 300px) {
14        #sidebar {
15          display: none;
16        }
17      }
18 </style>
```

“,”: logical or

dedicated CSS files

rules for different devices in one file

when printing, use black and white

“and”: logical and

hide the sidebar for small devices

Animations and transitions

In general …

- CSS styles (states) are defined by the user, the rendering engine takes care of the transition between styles

- **Animations** consist of:
  - an animation style (linear, etc.)
  - a number of “keyframes” that act as transition waypoints

- **Transitions** are animations:
  - that consist of exactly 2 states: start and end state
  - (have an alternative simple syntax)

CSS vs. JavaScript animations

- Easy to use (standard CSS) — no need to learn JavaScript

- Rendering engines are optimised for CSS-based animations

- CSS animations can do much more than animating buttons

CSS animation example (Firefox)

```
1  #p1 {
2      animation-duration: 5s;
3      animation-name: myname;
4      top: 5px; left: 5px;
5  }
6
7  @keyframes myname {
8      from {
9          top:5px; left:5px;
10         background-color: lightgreen;
11     }
12     50% {
13         background-color: red;
14     }
15     to {
16         top:5px; left:250px;
17         background-color: lightblue;
18     }
19 }
```

duration of animation (seconds)

animation name (@keyframes)

start state

intermediate state

end state

http://jsfiddle.net/Lyh5qvfo/1/

CSS animation example (-webkit-)

```
 1 #p1 {
 2     -webkit-animation-duration: 5s;
 3     -webkit-animation-name: pToRight;
 4     top: 5px; left: 5px;
 5 }
 6
 7 @-webkit-keyframes pToRight {
 8     from {
 9         top:5px; left:5px;
10         background-color: lightgreen;
11     }
12     50% {
13         background-color: red;
14     }
15     to {
16         top:5px; left:250px;
17         background-color: lightblue;
18     }
19 }
```

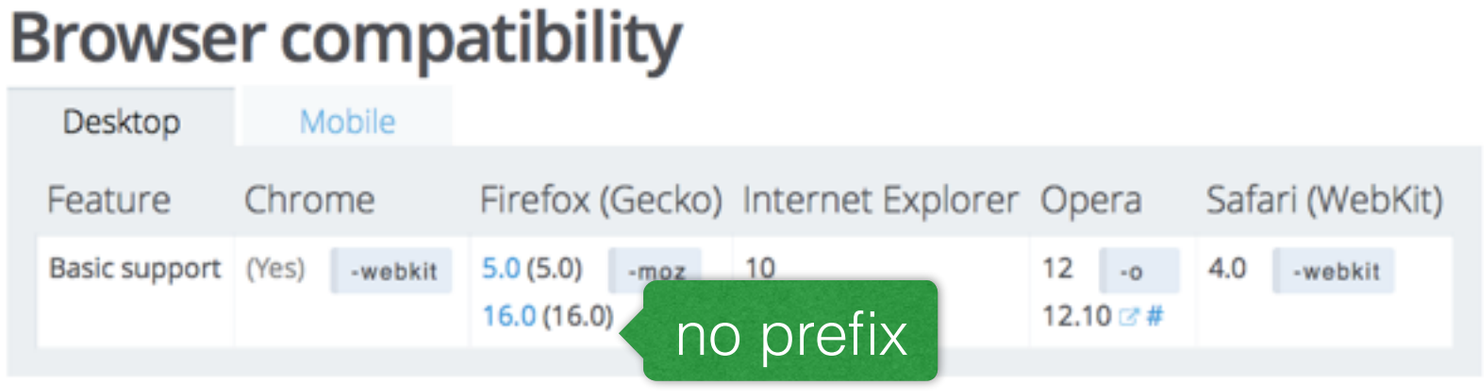
to support diff. browsers, the code needs to be repeated for every browser prefix

CSS animation control

`animation-iteration-count`
number of times an animation is executed (default: 1); value either a positive number or **`infinite`**

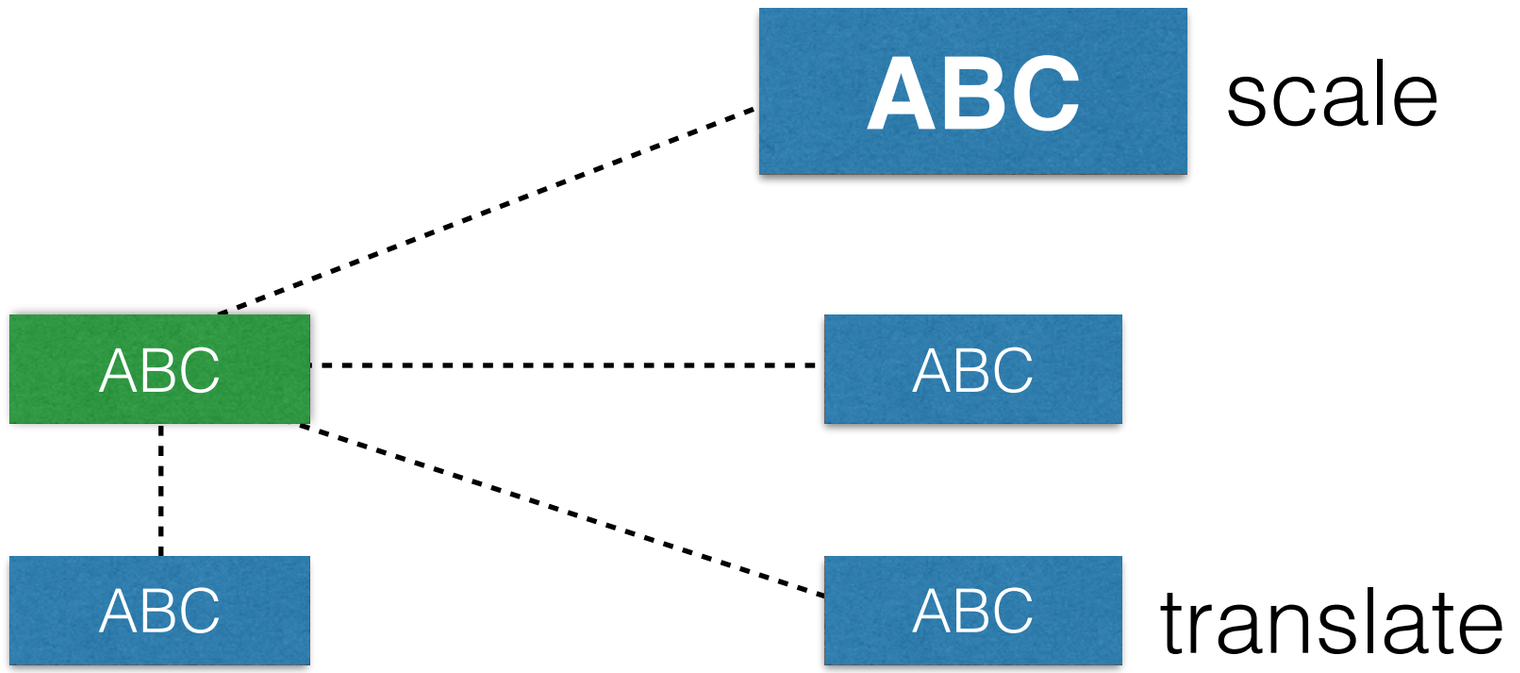
`animation-direction`
by default the animation restarts at the starting keyframe; if set to **`alternate`** the animation direction change every iteration

`animation-delay`
number of seconds until the animation starts (default 0s)

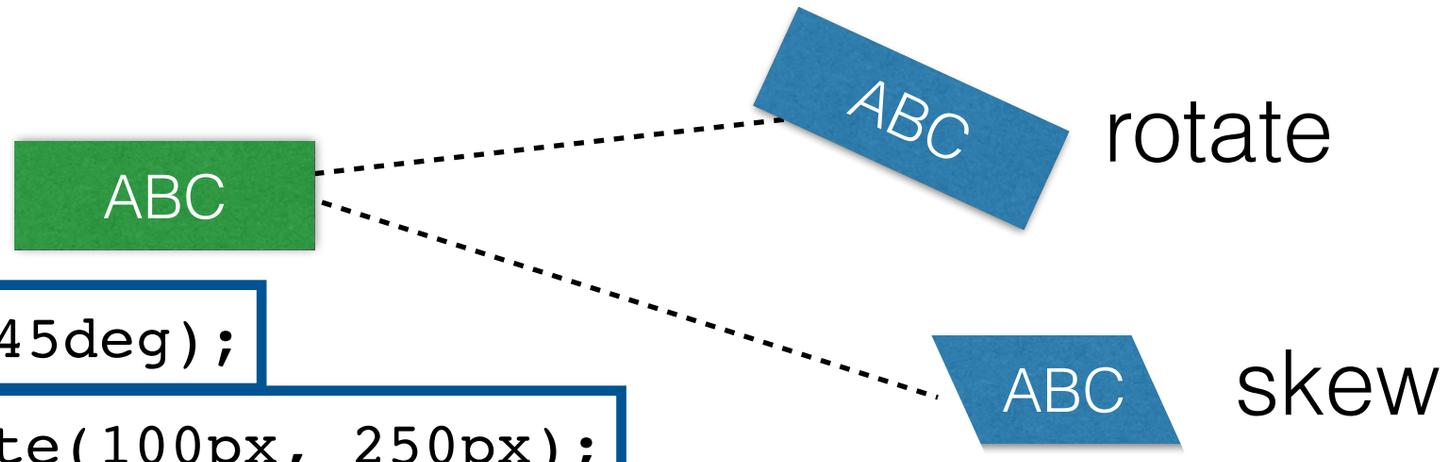


`transform`

So far we can:



With `transform` we can also:




```
1 transform: rotate(45deg);
2 transform: translate(100px, 250px);
3 transform: translate(10px,10px) skew(20deg);
```

http://jsfiddle.net/67tdgve7/

CSS transitions

```
1  .box {
2      border-style: solid;
3      border-width: 1px;
4      display: block;
5      width: 100px;
6      height: 100px;
7      background-color: red;
8      -webkit-transition:width 2s, height 2s, -webkit-transform 2s;
9      transition:width 2s, height 2s, transform 2s;
10 }
11 .box:hover {
12     background-color: green;
13     width:200px;
14     height:200px;
15     -webkit-transform:rotate(180deg);
16     transform:rotate(180deg);
17 }
```

start state

declare transition

end state

We have been using (default) transitions all the time.

CSS3 can do so much more!

http://neography.com/experiment/circles/solarsystem/#sun

https://codepen.io/juliangarnier/pen/idhuG

http://www.clicktorelease.com/code/css3dclouds/

Ajax

Dynamic updating on the client

Development strategy

- Develop the **client-side code** (HTML, CSS, JavaScript)
- Place all files into some directory (e.g. `/client`) **on the server**
- Define the **node.js server code** in a *.js file using Express
- Set **Express' static file path** to the directory of step 2

- **Add interactivity between client and server via Ajax and JSON**

```
server.js
client/
    html/
        =>index.html
        =>error.html
    images/
        =>background.png
        =>logo.png
css/
    =>layout.css
    =>style.css
javascript/
    =>todos.js
```

On the server: sending JSON

```
1  var express = require("express");
2  var url = require("url");
3  var http = require("http");
4  var app;
5
6  var port = process.argv[2];
7  app = express();
8  http.createServer(app).listen(port);
9
10 var todos = {};
11 var t1 = { message : "Maths homework due", type
12           : 1, deadline : "12/12/2014"};
13 var t2 = { message : "English homework due",
14           type : 3, deadline : "20/12/2014"};
15 todos[31212] = t1;
16 todos[23232] = t2;
17
18 app.get("/todos", function (req, res) {
19  res.json(todos);
20 });
```

On the server: sending JSON

```
1  var express = require("express");
2  var url = require("url");
3  var http = require("http");
4  var app;
5
6  var port = process.argv[2];
7  app = express();
8  http.createServer(app).listen(port);
9
10 var todos = {};
11 var t1 = { message : "Maths homework due", type
12           : 1, deadline : "12/12/2014"};
13 var t2 = { message : "English homework due",
14           type : 3, deadline : "20/12/2014"};
15                    t1;
16                    t2;
17
18 app.get("/todos", function (req, res) {
19  res.json(todos);
20 });
```

we store all TODOs on the server

Client requests the server's "copy" of the TODOs.

the client is sent the JSON formatted todos

On the client: basic HTML

jQuery way

```
1  <!doctype html>
2  <html>
3      <head>
4          <title>Plain text TODOs</title>
5          <script src="http://code.jquery.
6                       com/jquery-1.11.1.js"
7                       type="text/javascript"></script>
9          <script src="client-app.js"
10                      type="text/javascript"></script>
12     </head>
13     <body>
14       <main>
15         <section id="todo-section">
16           <p>My list of TODOS:</p>
17           <ul id="todo-list">
18           </ul>
19         </section>
20       </main>
21     </body>
22 </html>
```

Load the JavaScript files, start with jQuery

Define where the TODOs will be added.

On the client: JavaScript

jQuery way

```
1  var main = function () {
2   "use strict";
3
4   var addTodosToList = function (todos) {
5      var todolist = document.getElementById("todo-
6                                  list");
7      for (var key in todos) {
8           var li = document.createElement("li");
9           li.innerHTML = "TODO: “+todos[i].message;
10          todolist.appendChild(li);
11     }
12  };
13
14  $.getJSON("todos", addTodosToList);
15 }
16 $(document).ready(main);
```

On the client: JavaScript

jQuery way

```
1  var main = function () {
2   "use strict";
3
4   var addTodosToList = function (todos) {
5      var todolist = document.getElementById("todo-
6                                          list");
7      for (var key in todos) {
8         var li = document.createElement("li");
9         li.innerHTML = "TODO: “+todos[i].message;
10        todolist.appendChild(li);
11     }
12  };
13
14  $.getJSON("todos", addTodosToList);
15 }
16 $(document).ready(main);
```

Define what happens when a todo object is available (callback function)

Dynamic insert of list elements into the DOM

this is Ajax

when the call to /todos has returned, execute addTodosToList

when the document is loaded, execute main()

Ajax

only in the name

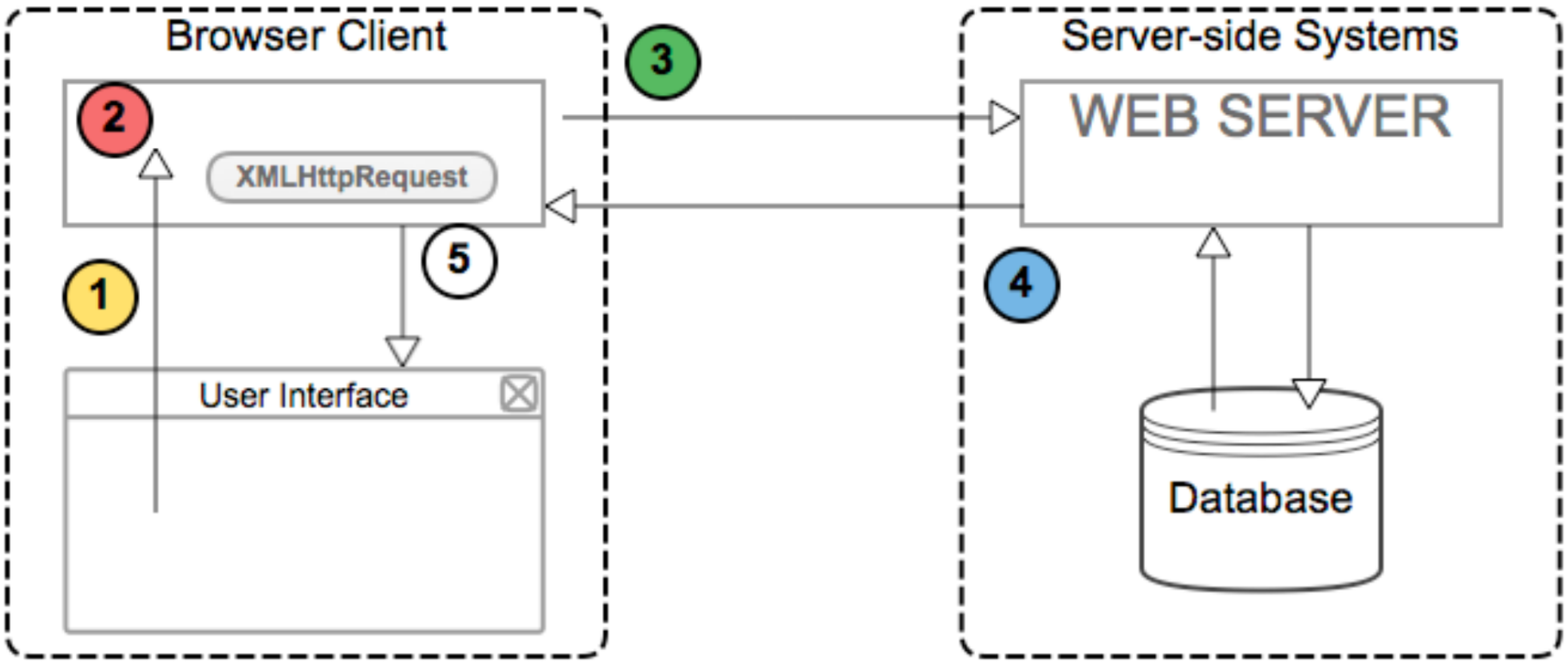
Asynchronous JavaScript and XML

- Ajax is a **JavaScript mechanism** that enables the dynamic loading of content without having to refetch/ reload the page manually
- Ajax: technology to inject new data into an existing web page (not a language or a product)
- You see this technology every day: chats, endless scrolling
- Ajax revolves around `XMLHttpRequest`, a JavaScript object
- **jQuery hides all complexity, makes Ajax calls easy**

Ajax: how does it work?

1. Web browser creates a **`XMLHttpRequest`** object
2. **`XMLHttpRequest`** requests data from a Web server
3. Data is sent back from the server
4. On the client, JavaScript code injects the data into the page

Ajax: how does it work?



1. JavaScript call
2. XMLHttpRequest
3. HTTP request
4. Data returns
5. HTML & CSS data

Without Ajax …

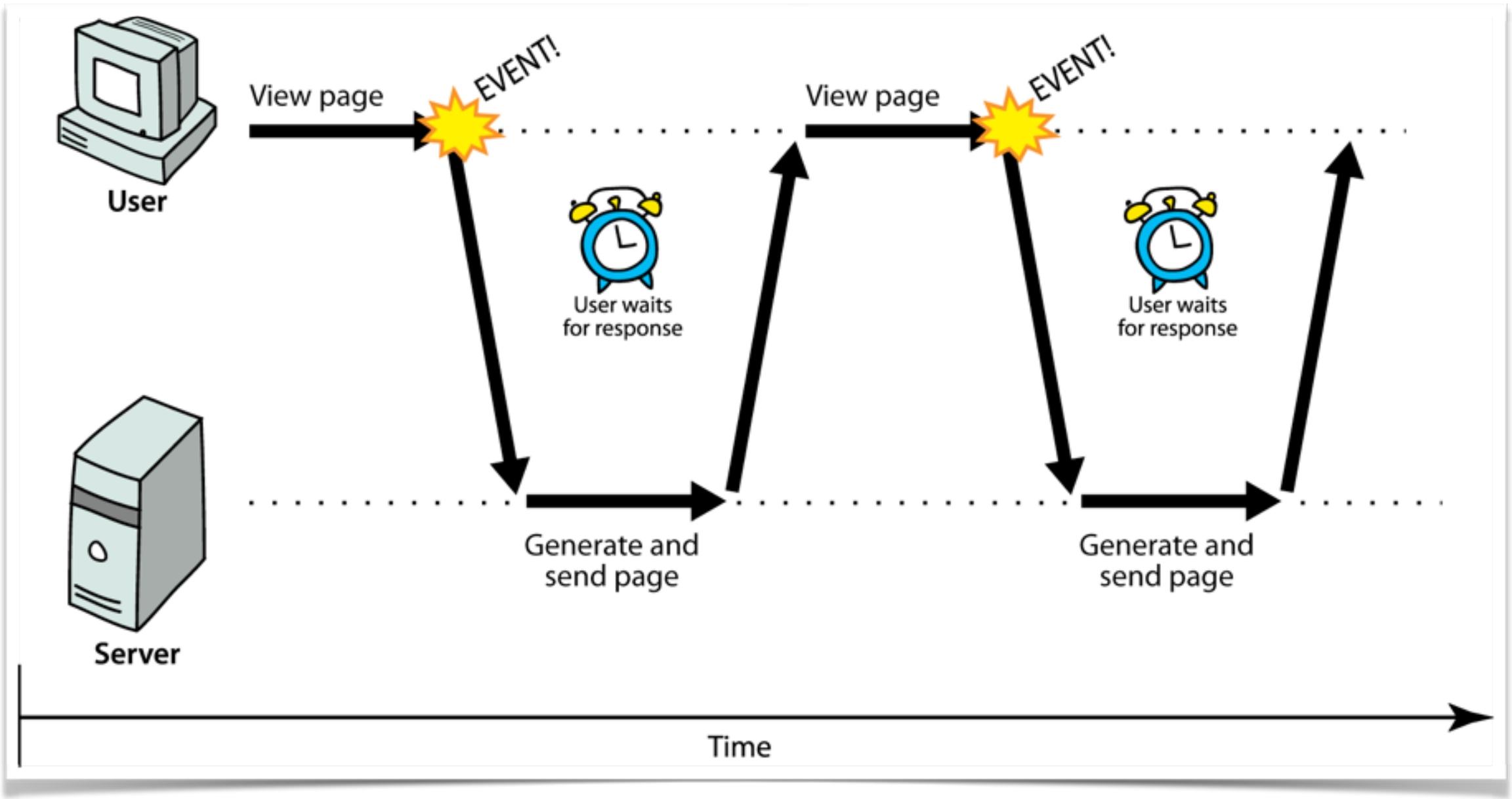


Image source: Web Programming Step by Step

Ajax works differently

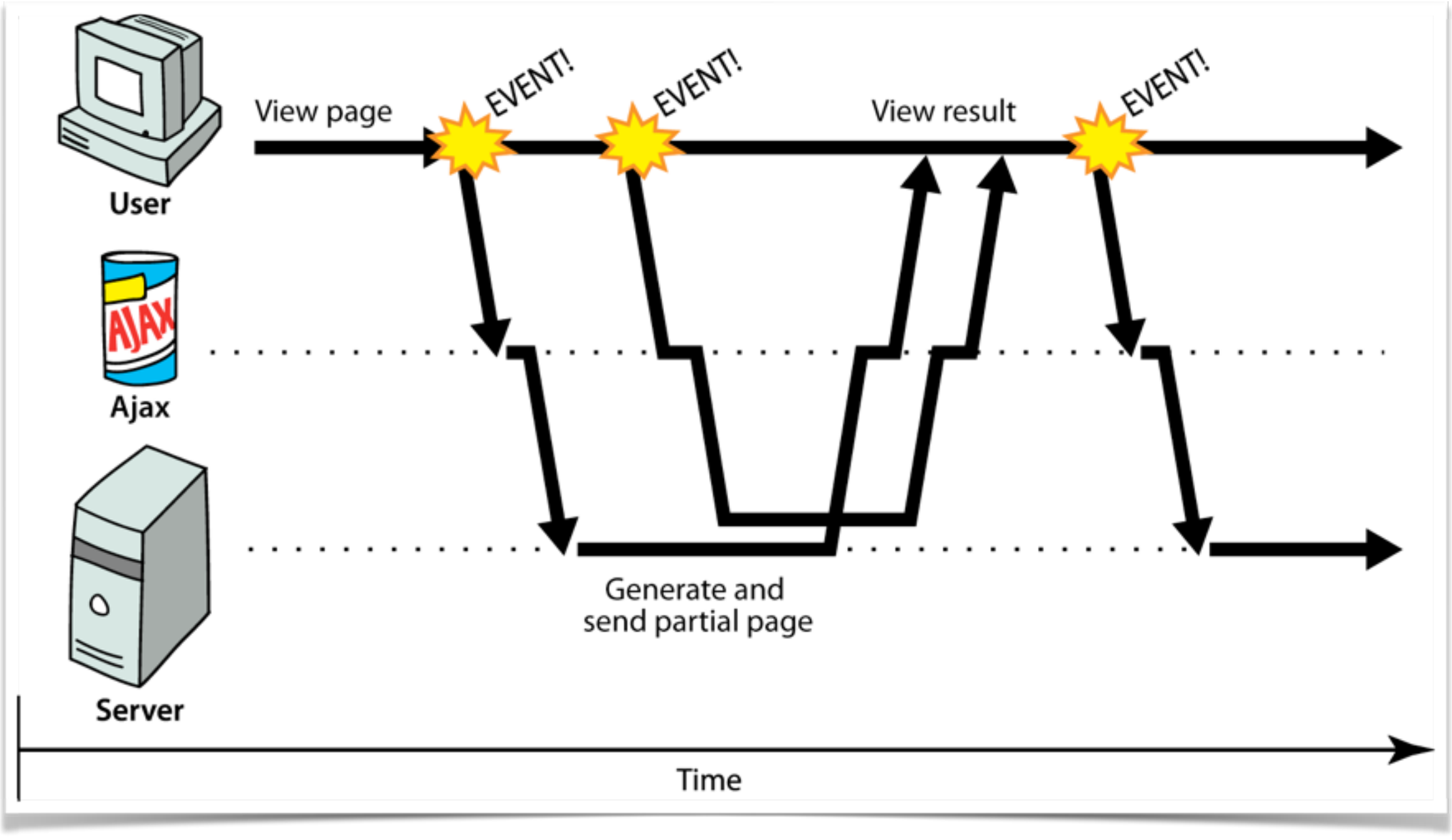


Image source: Web Programming Step by Step

Ajax: synchronous request

plain JS

```
1  //IE6 and prior IE versions use Microsoft.
2  XMLHTTP instead
3  var ajax = new XMLHttpRequest();
4
5  //retrieve data from URL (file) of interest
6  //false parameter: synchronous request
7  ajax.open('GET', 'example.txt', false);
8  ajax.send(null);
9
10 //response data in ajax.responseText
11 document.getElementById('ttExampleText').value
12                                  = ajax.responseText;
```

line of code is executed after line 7/8 return.

Ajax: XMLHttpRequest

plain JS

property	description
`onreadystatechange`	fires when status changes
`readyState`	0: not initialized 1: set up, not sent 2: sent 3: in progress 4: completed
`responseText`	data that was fetched (string)
`responseXML`	fetched XML as DOM `Document` object
`status`	HTTP status code sent back by the Web server (e.g. 200, 403)
`statusText`	description of status sent by the Web server

Ajax: XMLHttpRequest

plain JS

property	description
`onreadystatechange`	fires when status changes
`readyState`	0: not initialized 1: set up, not sent 2: sent 3: in progress 4: completed
`responseText`	data that was fetched (string)
`responseXML`	fetched XML as DOM `Document` obje
`status`	HTTP the V
`statusText`	desc Web

method	description
`abort()`	stop a Web request in progress
`open(method, url, async)`	GET or POST URL to fetch true (asynchronous) or false (synchronous)
`send(data)`	called after `open()`; for POST `data` is the string to send, for GET it is `null`

Ajax: an asynchronous request

plain JS

```
1 var ajax = new XMLHttpRequest();
2
3 //function to be called when the data has
4 arrived
5 ajax.onreadystatechange = function() {
6
7     //the only state we care about
8     if(ajax.readyState==4) {
9         /*
10          * process received data
11          */
12    }
13 }; //end of function
14
15 ajax.open("GET", "url", true); //true indicates
16           asynchronous request
17
18 ajax.send(null);
```

Ajax: an asynchronous request

plain JS

```
1  var ajax = new XMLHttpRequest();
2
3  //function to be called when the data has
4  arrived
5  ajax.onreadystatechange = function() {
6
7     //the only state
8     if(ajax.readyStat
9        /*
10        * process received data
11        */
12    }
13 }; //end of function
14
15 ajax.open("GET", "url", true); //true indicates
16           asynchronous request
17
18 ajax.send(null);
```

Event onreadystatechange is fired when the status of the request changes.

Ajax security

- Conveniently we always requested data from "our" Web server
- **Security restriction of Ajax**: can only fetch files from the same Web server as the calling page (*Same-origin policy*)
  - Same origin when protocol, port, and host are the same for two pages
- Ajax **cannot** be executed from a Web page stored locally on disk

the course book explains how to get around this (not recommended)

Node.js

Organisation and reusability of node.js code

So far

- All server-side code maintained within a single file
- Possible for small projects
- Larger projects suffer in this setting
  - Debugging is cumbersome
  - Team-work is cumbersome
  - Programming is cumbersome

node.js modules

- Code can be organised in **modules**
- Not all functions and variables in a module are exposed to the application
  - Exposed elements are made known via **`exports`**
- Modules can be **published** to the **npm**
  - Distribution of modules to other developers is made easy

node.js modules

- Code can be organised in **modules**
- Not all functions and variables in a module are exposed to the applicatior
  - Exposed ele
- Modules can b
  - Distribution easy

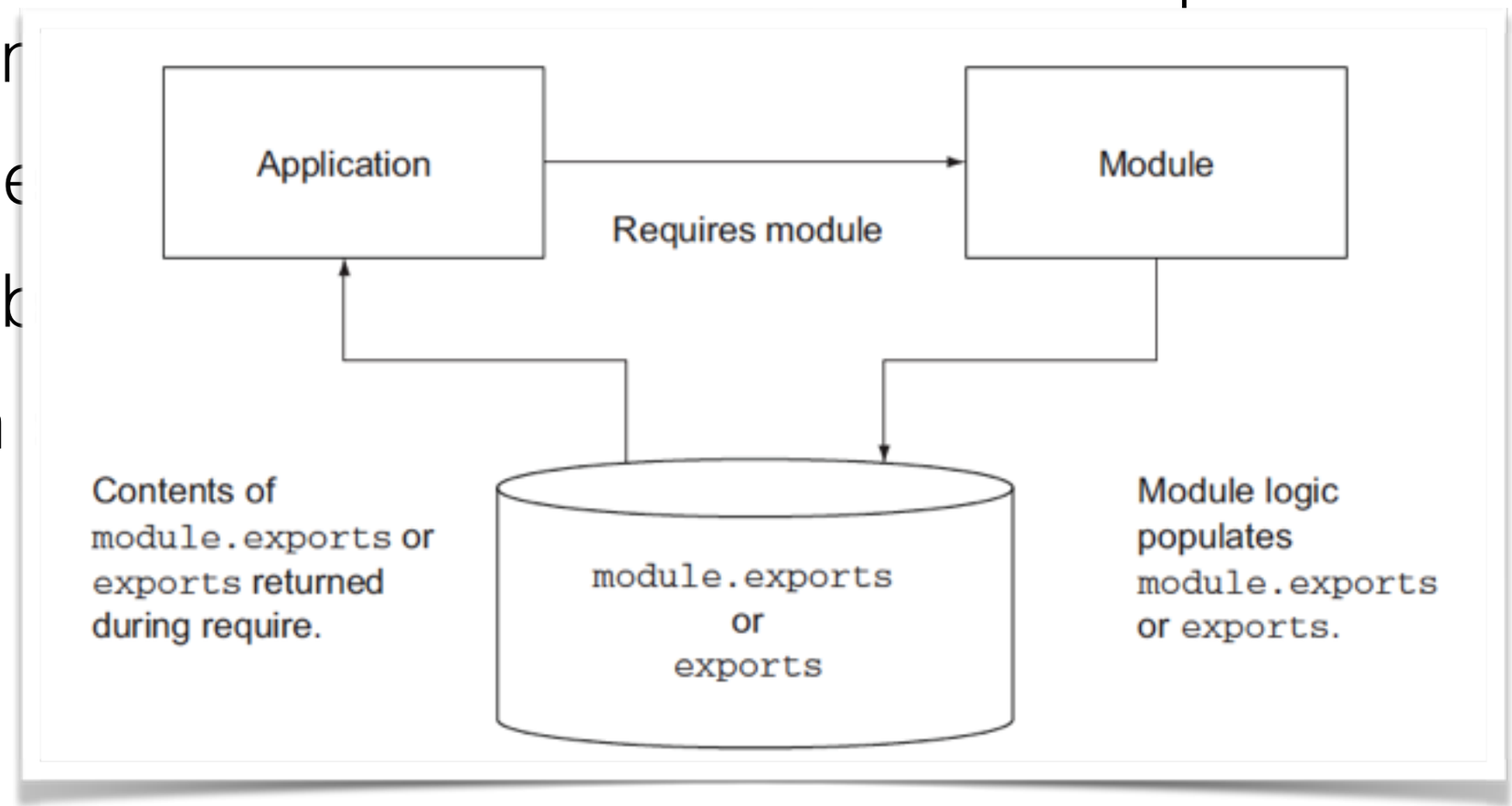


Image source: Node.js in Action

Creating a module

- A module can be
  - a single file, or
  - a directory of files (which includes a file index.js)

```
 1 function roundGrade(grade) {
 2     return Math.round(grade);
 3 }
 4
 5 function roundGradeUp(grade) {
 6     return Math.round(0.5+parseFloat(grade));
 7 }
 8 exports.maxGrade = 10;
 9 exports.roundGradeUp = roundGradeUp;
10 exports.roundGradeDown = function(grade) {
11     return Math.round(grade-0.5);
12 }
```

not exposed in this module; application cannot use it

grades.js

Using a module

require returns the contents of the exports object

```
1  var express = require("express");
2  var url = require("url");
3  var http = require("http");
4  var grading = require("./grades");
5  var app;
6
7  var port = process.argv[2];
8  app = express();
9  http.createServer(app).listen(port);
10
11 app.get("/round", function (req, res) {
12   var query = url.parse(req.url, true).query;
13   var grade = ( query["grade"]!=undefined) ?
14                 query["grade"] : "0";
15   res.send("Rounding up: " +
16            grading.roundGradeUp(grade) +", and down: "+
17            grading.roundGradeDown(grade));
18 });
```

adding our module (current directory)

accessing module functions

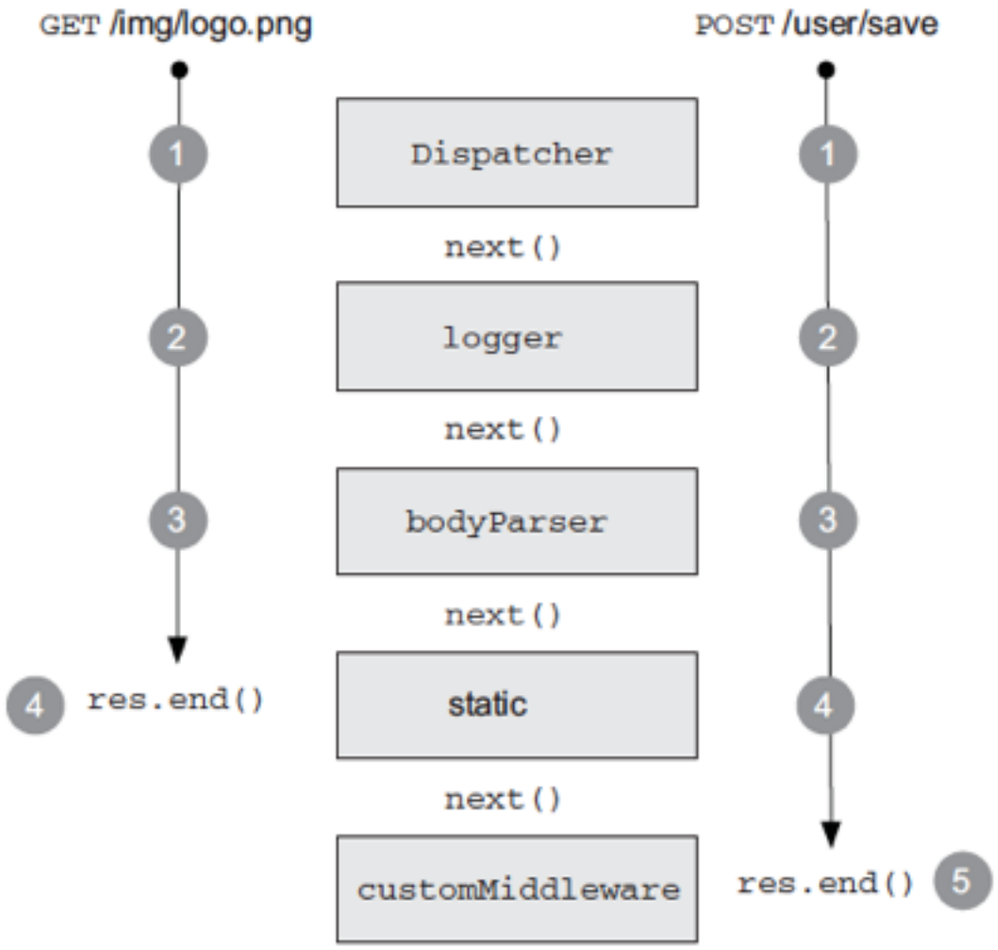
Express and Connect

What is Connect?

Express is actually built on top of Connect

- **Connect**: **framework** whose components ("**middleware**") are used to create Web application logics (inspired by Ruby's Rack framework)
- Middleware: **function** which **intercepts** the HTTP server's request & response objects, **executes logic** and ends the response or passes them to the next component
- **Dispatcher** connects the components (thus "Connect")
- Typical components: request logging, static file serving, request body parsing, session managing, etc.

Connect example



1. Dispatcher receives request and passes it to the first middleware
2. Request is logged and passed to the next middleware using next()
3. Request body is parsed if any exists and then passed to the next middleware using next()
4. If request is for a static file, response is sent with that file and next() is not called; otherwise, the request moves to the next middleware
5. Request is handled with a custom middleware and response is ended

Image source: *Node.js in Action*, page 124

A minimal Connect example

```
1 var connect = require('connect');
2 var app = connect();
3 app.listen(3000);
```

You should recognise this setup from Express

- Example has no middleware attached
- Dispatcher invokes each middleware component until one responds
  - If none responds (or none exists), the dispatcher will eventually send back a 404 response to the client

Middleware components

- Middleware components take three arguments:
  - HTTP request object
  - HTTP response object
    - Callback function (`next()`) which indicates that the component is finished and the dispatcher can move to the next component
- Middleware components are **small**, **self-contained** and **reusable** across applications

A simple logger component

- **Goal**: create a log file which records the request method and the URL of the request coming into the server

- **Required**: JavaScript function which accepts the request and response objects and the next() callback function

```
 1 var connect = require('connect');
 2
 3 //a middleware logger component
 4 function logger(request, response, next) {
 5     console.log('%s\t%s\t%s', new Date(),
 6                 request.method, request.url);
 7     next();
 8 }
 9 var app = connect();
10 app.use(logger);
11 app.listen(3001);
```

A simple logger component

- **Goal**: create a log file which records the request method and the URL of the request coming into the server

- **Required**: JavaScript function which accepts the request and response objects and the next() callback function

```
 1 var connect = require('connect');
 2
 3 //a middleware logger component
 4 function logger(request, response, next) {
 5     console.log('%s\t%s\t%s', new Date(),
 6                     request.method, request.url);
 7     next();
 8 }
 9 var app = connect();
10 app.use(logger);
11 app.listen(3001);
```

control returns to the dispatcher

register middleware component

Lets add a component that responds to the client

- **Goal**: reply “Hello World” to any request
- **Required**: JavaScript function which accepts the request and response objects and the next() callback function

```
1  var connect = require('connect');
2
3  function logger(request, response, next) { .... }
4
5  function helloWorld(request, response, next) {
6      response.setHeader('Content-Type',
7                         'text/plain');
8      response.end('Hello World!');
9  }
10
11 var app = connect();
12 app.use(logger);
13 app.use(helloWorld);
14 app.listen(3001);
```

No call to next! Control is not returned to dispatcher

any number of components can be registered

Question: What is the point of this code?

```
1 var connect = require('connect');
2 connect()
3     .use(logger)
4     .use('/admin',restrict)
5     .use(serveStaticFiles)
6     .use(hello);
```

```
 1 function restrict(req, res, next) {
 2     var auth = req.headers.authorization;
 3     if (!auth)
 4         return next(new Error('Unauthorized'));
 5
 6     var parts = auth.split(' ');
 7     var scheme = parts[0];
 8     var auth = new Buffer(parts[1],
 9                     'base64').toString().split(':');
10     var user = auth[0]
11     var pass = auth[1];
12     if(user == "user" && pass == "password") { next(); }
13 }
```

Question: What is the point of this code?

connect4.js

```
var connect = require('connect');
connect()
    .use(logger)
    .use('/admin',restrict)
    .use(se...eStaticFiles)
    ...o);
```

every call to use() returns the connect object itself - allows method chaining.

middleware component is only invoked if the URL prefix matches

```
    ...rict(req, res, next) {
    var auth = req.headers.authorization;
```

```
curl --user user:password http://localhost:3001/admin
```

```
    var scheme = parts[0];
    var auth = new Buffer(parts[1],
                    'base64').toString().split(':');
    var user = auth[0]
    var pass = auth[1];
    if(user == "user" && pass == "password") { next(); }
}
```

Making the components configurable

- **Goal**: middleware components should be reusable across applications without additional engineering effort
- Approach: wrap original middleware function in a setup function which takes the parameters as input

```
1 function setup(options) {
2     // setup logic
3     return function(req, res, next) {
4         // middleware logic
5     }
6 }
7
8 app.use( setup({ param1 : 'value1' }) );
```

important: function CALL is made!

Templating

with EJS

Express and HTML …

```
1  var express = require("express");
2  var url = require("url");
3  var http = require("http");
4  var app;
5
6  var port = process.argv[2];
7  app = express();
8  http.createServer(app).listen(port);
9
10 app.get("/greetme", function (req, res) {
11   var query = url.parse(req.url, true).query;
12   var name = ( query["name"]!=undefined) ? query[
13                "name"] : "Anonymous";
14   res.send("<html><head></head><body><h1>
15             Greetings "+name+"</h1></body></html>
16             ");
17 });
18
19 app.get("/goodbye", function (req, res) {
20   res.send("Goodbye you!");
21 });
```

Express and HTML …

```
1  var express = require("express");
2  var url = require("url");
3  var http = require("http");
4  var app;
5
6  var port = process.argv[2];
7  app = express();
8  http.createServer(app).listen(port);
9
10 app.get("/greetme", function (
11   var query = url.parse(req.url
12   var name = ( query["name"]!=u
13                "name"] : "Anonymous";
14   res.send("<html><head></head><body><h1>
15             Greetings "+name+"</h1></body></html>
16             ");
17 });
18
19 app.get("/goodbye", function (req, res) {
20   res.send("Goodbye you!");
21 });
```

error-prone, ugly, not maintainable, fails at anything larger than a toy project.

Instead … templates

- **Goal:** write as little HTML “by hand” as possible

- Keeps the code clean - separates logic from presentation markup
- Many different template engines exist for node.js
- We focus first on **EJS** (*Embedded JavaScript*)**, a template engine and language**

> “EJS cleans the HTML out of your JavaScript with client side templates. After EJS gets its rubber gloves on dirty code, you'll feel organized and uncluttered.” — http://www.embeddedjs.com/

Model-View-Controller (MVC)

- Standard design pattern of how to keep **logic**, data and **presentation** separate
- User request of a resource from the server triggers …

1. **controller** requests application data from the **model**
2. **controller** passes the data to the view
3. **view** formats the data for the user (template engines are used here)

Model-View-Controller (MVC)

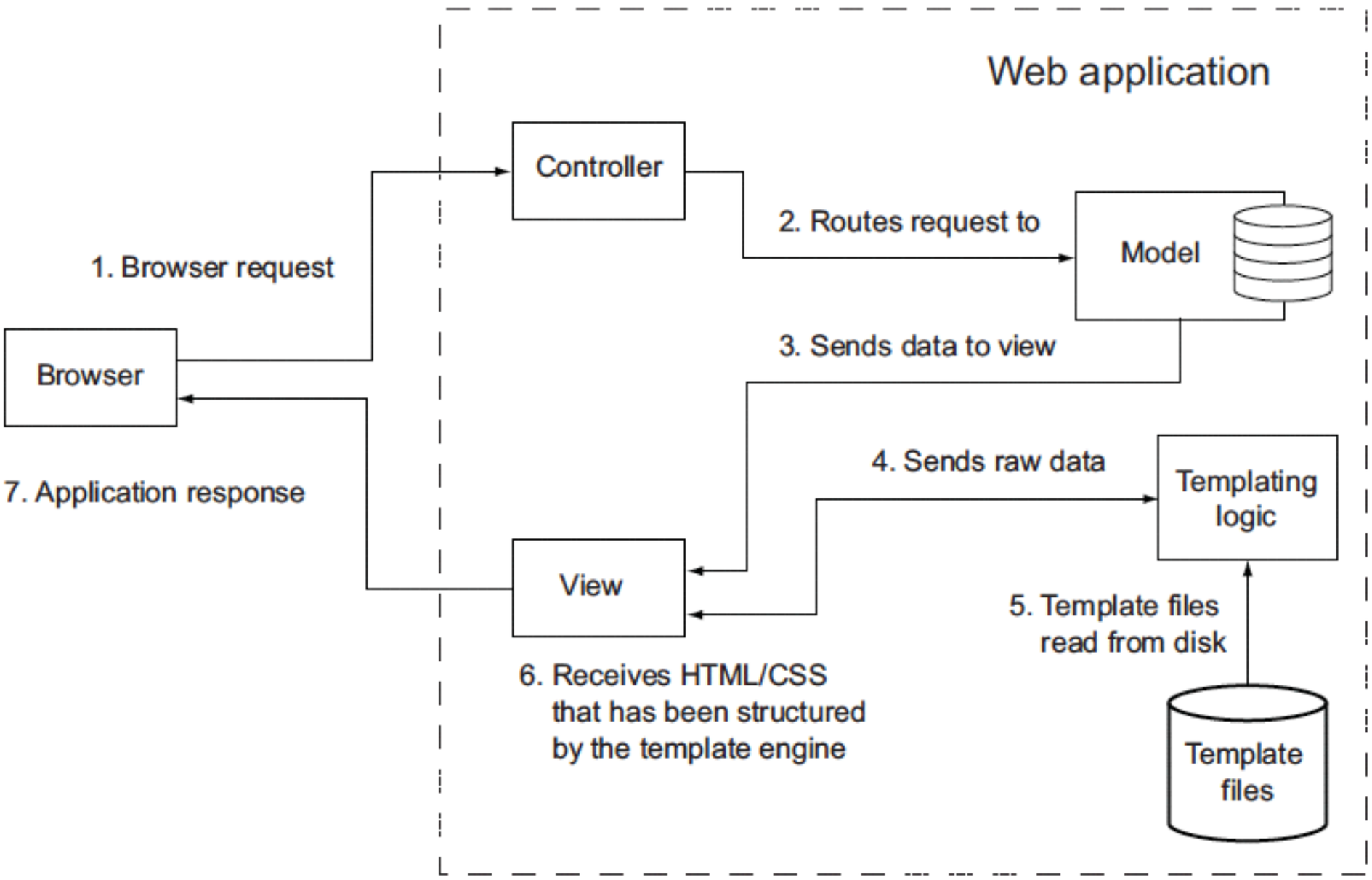


Image source: *Node.js in Action*, page 265

A first look at ejs

test-ejs.js

```
1 var ejs = require('ejs');
2 var template = '<%= message %>';
3 var context = {message: 'Hello template!'};
4 console.log(ejs.render(template, context));
```

Start on the console: node test-ejs.js

test-ejs2.js

using <%- instead if you do not want escaping
(enables cross-site scripting attack!)

```
1 var ejs = require('ejs');
2 var template = '<%= message %>';
3 var context = {message: "<script>alert('
4                 hi!');</script>"};
5 console.log(ejs.render(template, context));
```

By default, ejs escapes special values in the context.

ejs filtering

- Light-weight filtering can be defined within the template

```
1 var ejs = require('ejs');
2 var template = '<%=: movies | last %>';
3 var context = {'movies': [
4           'The Hobbit',
5           'X-Men',
6           'Superman V'
7 ]};
8 console.log(ejs.render(template, context));
```

ejs filtering

- Light-weight filtering can be defined within the template

```
1 var ejs = require('ejs');
2 var template = '<%=: movies | last %>';
3 var context = {'
4           'The Hob
5           'X-Men',
6           'Superma
7 }};
8 console.log(ejs.render(template, context));
```

: indicates filtering

filter is defined in the template. Only select the last element of an array.

- Filtering based on array position:
  - `last` to select the last item
  - `first` to select the first item
  - `get:n` to select the nth item (index starts at 0)

ejs filtering

- Numerous text manipulation filters exist
  - `upcase` to capitalize a word
  - `downcase` to lowercase a word
  - `truncate:n` limits the input to the first n characters
  - `truncate_words:n` limits the input to the first n words

```
1 var ejs = require('ejs');
2 var template = '<%=: title | truncate:16 %>';
3 var context = {title: 'I should keep it short
4                 but I cannot'};
5 console.log(ejs.render(template, context));
```

  - `replace` (either a string or a regular expression)

ejs filtering

- Filters can be chained

```
1 var ejs = require('ejs');
2 var template = '<%=: movies | sort | get:1 %>';
3 var context = {'movies': [
4           'The Hobbit',
5           'X-Men',
6           'Superman V'
7 ]};
8 console.log(ejs.render(template, context));
```

Chaining via the pipe symbol

ejs filtering

- You can define your own filters

round is not predefined

creating a filter

```
1  var ejs = require('ejs');
2  var template = '<%=: price * 1.145 | round:2 %>
3                 ';
4  var context = {price: 21};
5  ejs.filters.round = function(number,decimalPlaces) {
6
7      number = isNaN(number) ? 0 : number;
8      decimalPlaces = !decimalPlaces ? 0 :
9                      decimalPlaces;
10     var multiple = Math.pow(10, decimalPlaces);
11     return Math.round(number * multiple)/multiple;
12
13 };
14 console.log(ejs.render(template, context));
```

Configuring views with express

- Setting the “views” directory (i.e. the directory of the templates)

```
app.set('views', __dirname + '/views');
```

a global variable in Node

- Setting the template engine

```
app.set('view engine', 'ejs');
```

- Note: an application can make use of several template engines at the same time
- Numerous template engines exist for node.js

Exposing data to views

```
 1 var express = require("express");
 2 var url = require("url");
 3 var http = require("http");
 4 var app;
 5
 6 var port = process.argv[2];
 7 app = express();
 8 http.createServer(app).listen(port);
 9
10 var todos = [];
11 todos.push({ message: 'Midterm exam tomorrow',
12              dueDate: '12/11/2014'  });
13 todos.push({ message: 'Prepare for assignment
14              5', dueDate: '05/01/2015' });
15 todos.push({ message: 'Sign up for final exam',
16              dueDate: '06/01/2015' });
17
18
19 app.set('views', __dirname + '/views');
20 app.set('view engine', 'ejs');
21
22 app.get("/todos", function (req, res) {
23     res.render('todos', { title: 'My list of
24                 TODOs', todo_array: todos });
25 });
```

Exposing data to views

```
 1 var express = require("express");
 2 var url = require("url");
 3 var http = require("http");
 4 var app;
 5
 6 var port = process.argv[2];
 7 app = express();
 8 http.createServer(app).listen(port);
 9
10 var todos = [];
11 todos.push({ message: 'Midterm exam tomorrow',
12              dueDate: '12/11/2014'  });
13 todos.push({ message: 'Prepare for assignment
14              5', dueDate: '05/01/2015' });
15 todos.push({ message: 'Sign up for final exam',
16              dueDate: '06/01/2015' });
17
18
19 app.set(                                  iews');
20 app.set(
21
22 app.get( '/todos', function (req, res) {
23     res.render('todos', { title: 'My list of
24                           o_array: todos });
25 });
```

the list of todos we want to serve to the clients

`render()` indicates the use of a template

variables of the template

template to use

ejs template file

JavaScript between <% %> is **executed**.

JavaScript between <%= %> **adds output** to the result file.

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4   <title><%= title %></title>
5  </head>
6  <body>
7   <h1>TODOs</h1>
8   <div>
9      <% todo_array.forEach(function(todo) { %>
10     <div>
11        <h3><%=todo.dueDate%></h3>
12        <p><%=todo.message%></p>
13     </div>
14     <% }) %>
15  </div>
16 </body>
17 </html>
```

A last word on templates

- ejs still retains the original HTML tags
- Other template languages do not, **Jade** is a popular example here - which one to choose depends on your personal preferences

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) {
         bar(1 + 5)
      }
  body
    h1 Jade - node template engine
    #container.col
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Jade</title>
    <script type="text/javascript">
      if (foo) {
         bar(1 + 5)
      }
    </script>
  </head>
  <body>
    <h1>Jade - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
      </p>
    </div>
  </body>
</html>
```

Summary

- CSS: media queries and animations
- Ajax: how does it work behind the scenes (without jQuery)
- Templating with ejs

End of Lecture