COEN 168/268

Mobile Web Application Development

APIs and JSON

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University

Apps are nothing without data…

As an app developer you need to:

- Create a server-side source of data that persists
- Easily get that data to your app in a convenient format
- Be able to create, read, update, or delete data
- Also, be able to do this without reloading the page

As a result, you need to use the following acronyms:

- Use **REST** and **JSON** to create a web service **API**
- Implement **AJAX** calls on the client to send and receive data

What is REST, JSON, and AJAX?

- **REST:** The principles for creating a architecturally sound source of data through a defined web service
- **JSON:** The structure of the data that you're sending
- **AJAX:** The implementation of fetching and writing data in the client to the server

Now, let's talk about each one...

REST

REpresentational **S**tate **T**ransfer

What is REST?

- Created by Roy Fielding in 2000 in his Ph. D. thesis to describe a set of architectural principles for implementing the HTTP (Hypertext Transfer Protocol)
- In RESTful systems, servers expose resources using a URI (Uniform Resource Identifier, a URL is a form of a URI)
- These resources are accessed by clients (like the web browser) using the four HTTP resource verbs:
  - GET, POST, PUT, and DELETE

What is a resource?

A resource represents a tangle object that you can operate on

Basic REST Principles

- App state and functionality are divided into resources
- Resources can be addressed using standard URIs that can be used as hyperlinks
- Resources are only accessed using the the four HTTP verbs
- All resources provide information using the MIME types supported by HTTP
- The protocol is stateless, cacheable, and layered

A bit about the verbs used in HTTP requests

- **Use GET**: Used when getting a resource or a list of resources
- **Use DELETE:** Used when deleting an existing resource

Use POST:

- Use when you want to create a new resource for which client does not know the ID and you want the server to provide it
- Use it when you want to update a resource on the server that may require additional server-side work

Use PUT:

- Use it when you want to create a new resource and the client either assigns the ID for the resource or knows it already
- Use if when you want to update an existing resource by replacing it completely with the data that you send

These are also called CRUD operations (Create, Read, Update, Delete)

For POST and PUT you send data in the POST body of the request

For example, some REST actions...

GET `/todos/` - *retrieves a list of all todos*
GET `/todos/123` <- *retrieves the details of a resource with ID = 123*
PUT `/todos/123` <- *creates a new resource or fully updates an existing resource with ID = 123*
POST `/todos/new` <- *returns the id of the new resource that was created with the data provided*
POST `/todos/123` <- *updates an existing resource with ID = 123 but the server may update it additionally*
DELETE `/todos/123` <- *delegates the specified resource with ID = 123*

When designing your RESTful web service

- Provide **distinct** URIs for each resource you want to expose such as `/todos/` for to do items, or `/users/` for users
- Use nouns in the URIs (like `purchase`) do not make the URIs verbs (like `purchasing`) as actions are mapped through the HTTP methods
- **GET** calls should never change data on the server
- Make your service stateless because the client should not manage information state as your web service could be accessed by many clients

Some more RESTful URIs

- `/vehicles`
- `/vehicles/autos`
- `/vehicles/autos/{make}`
- `/vehicles/autos/{make}/{model}`
- `/vehicles/autos/{make}/{model}/{year}`

… and more

A note about RESTful URIs…

This is NOT RESTful:

```
/vehicles?type=autos&make=BMW&model=M3&year=2015
```

This IS RESTful:

```
/vehicles/autos/BMW/M3/2015
```

Not only is the RESTful URI easier to construct, but also human readable!

However, there are times when you need to add query parameters

- Not everything can be expressed through a URI
- Try to use the URI structure as much as possible
- However, for things like searches, you might want to get more power:

```
/vechicles/autos?sort=pricehigh&limit=20&offset=0
```

Returns a list of the first 20 automobiles sorted by price.

Constructing URLs

`<scheme>: <hierarchical> [?<query>] [#<fragment>]`

Specifically, the `<query>`:

- starts with ?
- parameters are of the format `key=value`
- separated with &
- need to be URI encoded

Okay, now we can construct a RESTful interface

Book inventory web service

- Let's say that you have a lot of books and you want a little app to keep track of all the books that you have
- You want to be able enter in all the books that you own
- Rank them by your rating
- Find books that you have read and ones that you haven't

What data does a book contain?

- `id` (string) primary key
- `title` (string)
- `author` (string)
- `publicationYear` (number)
- `rating` (number)
- `wasRead` (boolean)

Proposed endpoint: Getting all books with meta data

GET `/books`

- `limit` (number)
- `offset` (number)
- `sort` (string) "pubYear", "read", "unread", "author", "title", "purchaseDate"
- `filter` (string) "pubYear", "read", "unread", "author", "title", "purchaseDate"

Proposed endpoint: Getting data for one existing book

GET `/books/{id}`

- returns the book's data

Proposed endpoint: Editing an existing book

POST `/books/{uuid}`

- request body includes the data for the book, edited or not
- reflects the successfully saved book back

Proposed endpoint: Deleting an existing book

DELETE `/books/{id}`

- no request body
- just returns a 200 OK to reflect success

So, here' is our web service

GET `/books`
PUT `/books/new`
GET `/books/{id}`
POST `/books/{id}`
DELETE `/books/{id}`

Pretty simple!

What will we use to build this?

- PHP Flat File Database: https://github.com/wylst/fllat
    - `/books/index.php` <- list and book operations
- `.htaccess` for routing:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /books
  RewriteRule ^index\.php$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /books/index.php [L]
</IfModule>
```

Demo

Building the Book app's REST web service using PHP

Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/booksappdemo.zip

You can use have REST service return either XML or JSON

- XML is not used much anymore
- JSON, however, is dominant

JSON

What is JSON?

- Stands for **J**ava**S**cript **O**bject **N**otation
- Uses JavaScript literals to represent data
- It is much more lightweight than XML
- And as a bonus, JSON is just JavaScript so it is easy to get in and out of JavaScript-based apps

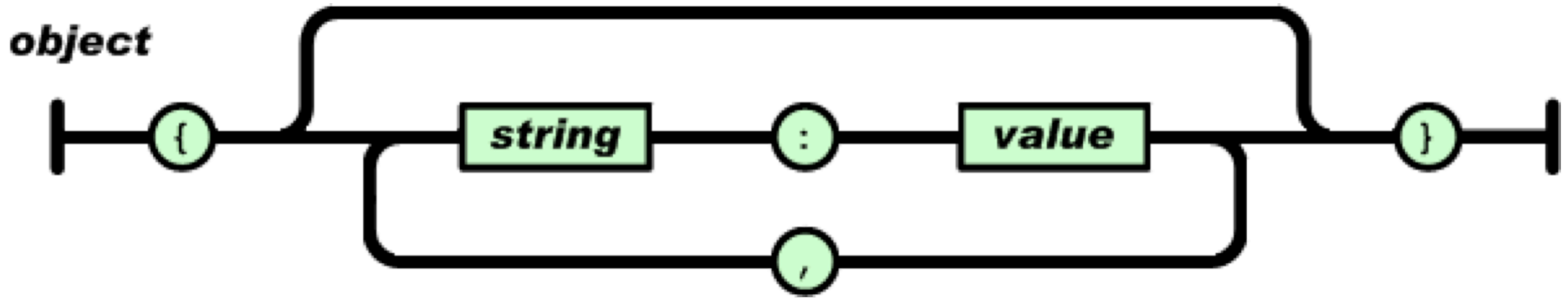
JSON is built on two structures:

- A collection of key/value pairs as objects.
- Ordered list of values as arrays.

The values in either of these can be objects or arrays to construct more complex structures.

Let's take a look at the definitions at JSON.org

JSON Object

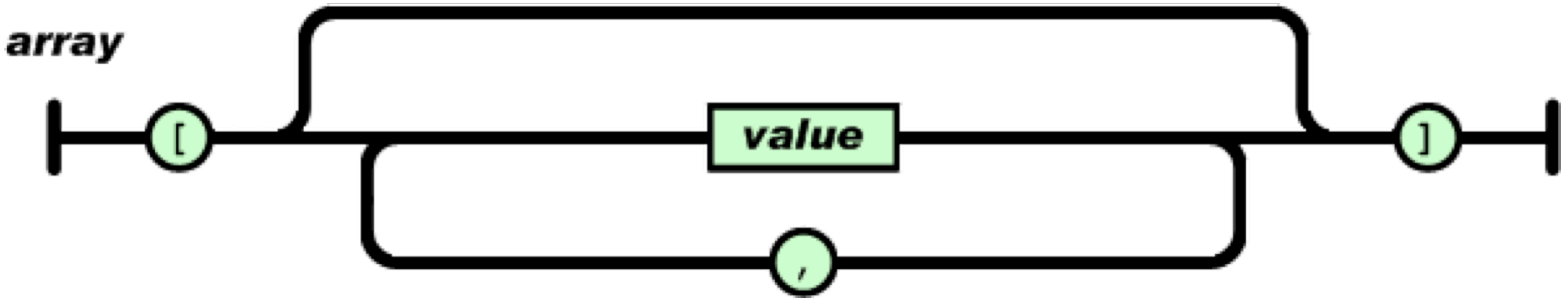


Values can be primitives or other objects

JSON Object

```
{
  "title": "APIs and JSON",
  "metadata": {
    "date": "2014-07-15T01:32:18-7:00",
    "duration": 2
  }
}
```

JSON Array

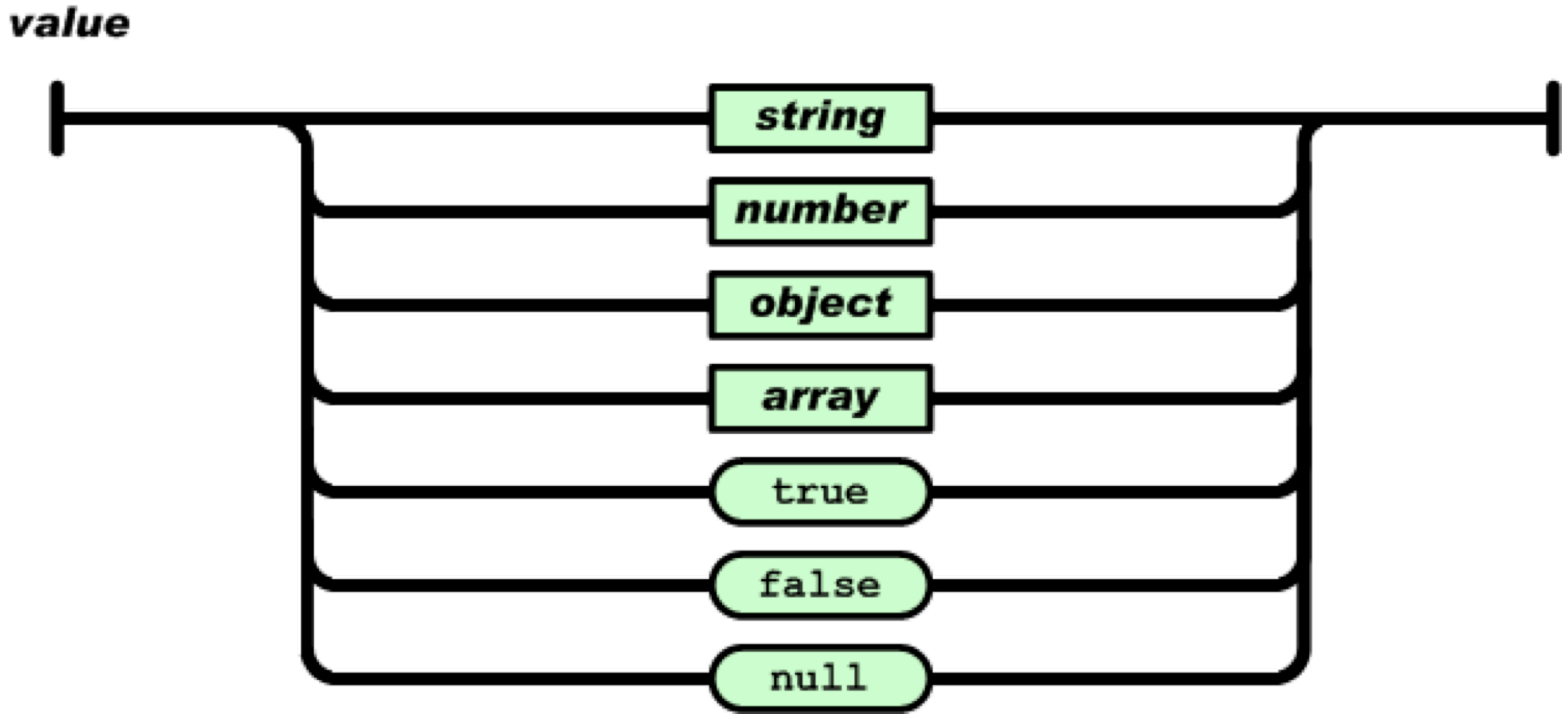


Values can be primitives or other objects

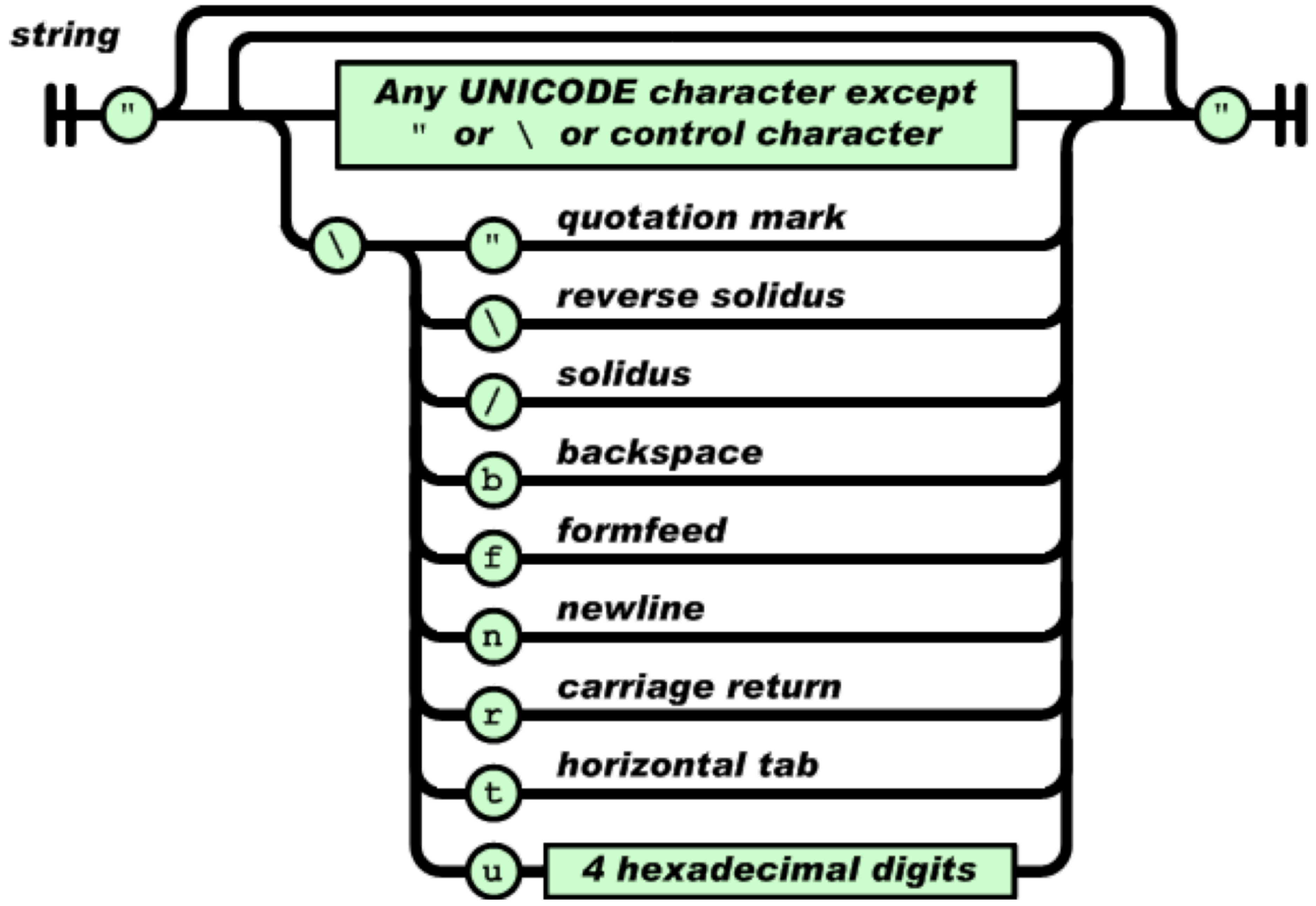
JSON Array

```json
{
  "colors": [
    {
      "hexValue": "#FF0000",
      "displayName": "Red"
    },
    {
      "hexValue": "#00FF00",
      "displayName": "Green"
    }
  ]
}
```

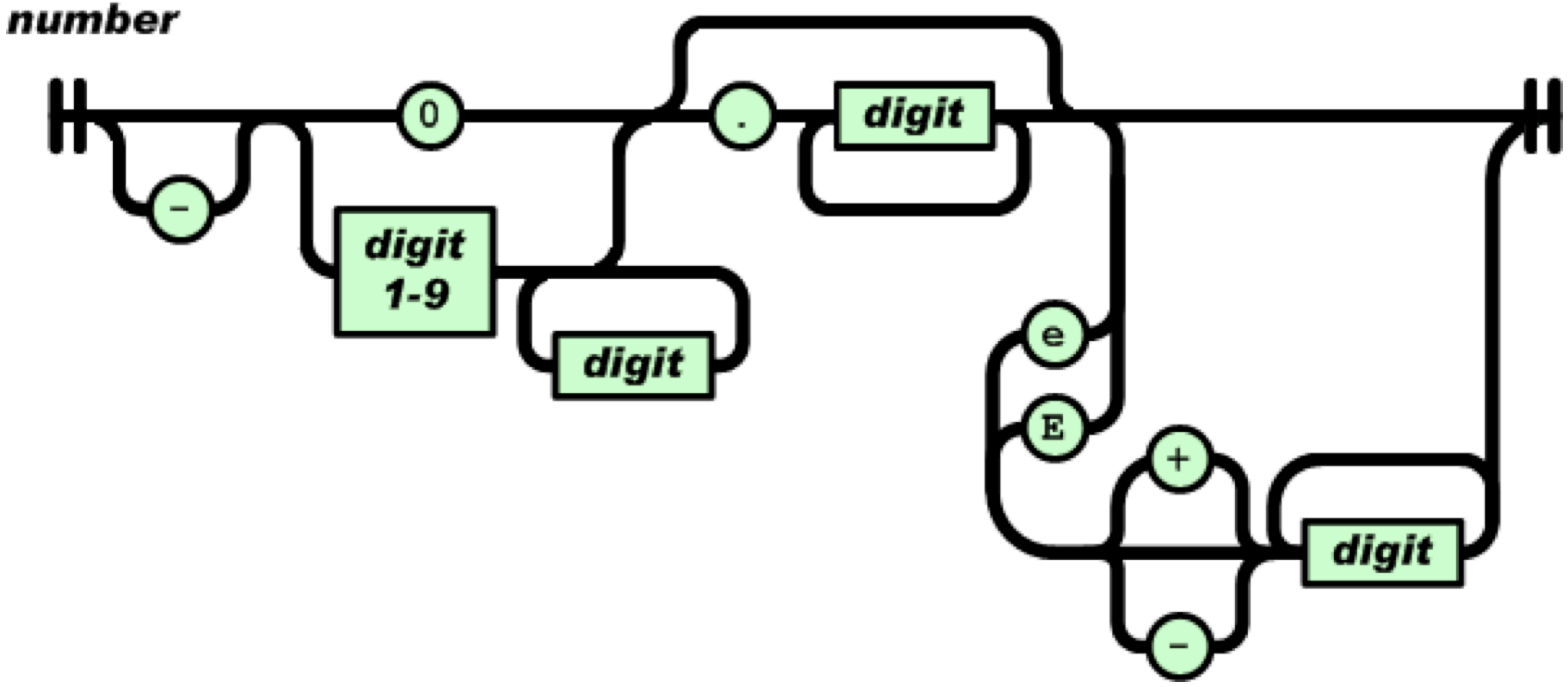
JSON Value



JSON String


string
"
Any UNICODE character except
" or \ or control character
\
"
quotation mark
\
reverse solidus
/
solidus
b
backspace
f
formfeed
n
newline
r
carriage return
t
horizontal tab
u
4 hexadecimal digits
"

JSON Number



So, let's see how we can define a JSON structure for the Books app

GET /books

```
{
  paginationInfo: {
    totalCount: {number}, // total number for query
    limit: {number}, // current limit returned
    offset: {number}, // current offset returned
  },
  books: [
    {book}, {book}, ...
  ]
}
```

What does a Book look like?

```
{
  "id":5,
   "title":"Harry Potter and the Chamber of Secrets",
   "isbn":"0-7475-3849-2",
   "pubYear":1998,
   "author":"J. K. Rowling",
   "rating":5,
   "wasRead":false,
   "note":"Need to read this one."
}
```

GET /books/5

```
{
  "id":5,
   "title":"Harry Potter and the Chamber of Secrets",
   "isbn":"0-7475-3849-2",
   "pubYear":1998,
   "author":"J. K. Rowling",
   "rating":5,
   "wasRead":false,
   "note":"Need to read this one."
}
```

Just returns the book's JSON

Now that we have both a REST API and a JSON structure

Let's get the data into the app!

AJAX

Asynchronous **J**avaScript **a**nd **X**ML

Before AJAX…

- Loading data from a server requires a page refresh
- Sending data to a server requires a page refresh
- This can be jarring to the user
- As web apps have become more complex, a lot of state remains in the browser
- This state would be lost if the page refreshes

With AJAX…

- Loading data from a server can be done in the background
- Sending data to a server can be done in the background
- This can be done without the user knowing
- Or you can put up UI indicating that a request is happening
- State remains in the browser as it doesn't reload

The XMLHttpRequest (XHR) Object

- Allows you to send and receive data without reloading the page
- Now, it is a standard object in all browsers
- XHRs can be asynchronous so that you can do other work on the page while waiting for data
- While XML is in the name, mostly JSON is used

Creating an XHR

```
var req = new XMLHttpRequest();
req.open('get', '/autos/bmw');

req.onreadystatechange = function() {
  if (req.readyState === 4) { // 4 means that the request is done
    if (req.status === 200) { // Success!
      alert(req.responseText);
    } else { // Failure
      alert('Error: '+req.status);
    }
  }
}

req.send(null);
```

However, with `$.ajax`, things are much simpler!

```
$.get('/autos/bmw', function(data) {
    alert(data);
});
```

You should use jQuery's `$.ajax` calls as they greatly simplify your code

Using jQuery's `$.ajax` is convenient

- Makes it easy perform HTTP methods
- Very easy to set up post body parameters
- Callbacks are easy to set up
- Easy to configure mime-types, headers, etc
- Built-in support for JSONP and cross domain requests

An example of callbacks

```
$.ajax('/autos/bmw/')
        .done(function() {
          console.log('success');

        }).fail(function() {
          console.log('failure');

        }).always(function() {
          console.log('completed');

        });
```

Saving data in a POST

```
$.ajax({
  type: 'POST',
  url: '/autos/,
  data: {
          model: 'Contour',
          make: 'Ford'
        }
}).done(function(result) {
  console.log('Saved: ' + result);
});
```

A note about cross domain requests

- You may have noticed all examples start with a `/` in their path
- That is because they are requests on the same `host` as the web page
- You cannot make requests from one domain to another due to security
- This is a pretty annoying limitation, but you learn to live with it

JSONP to the rescue!

- Instead of loading via XHR, the JSON is loaded with an external `<script>` tag
- There are no limitations with this method
- However, as it is loaded like a JavaScript file, you need to execute some code to get the data
- You specify `callBack` method in your code to be called from the data loaded in the `<script>` tag

With `$.ajax` it is done for you, just set the option

```
$.ajax({
  type: 'GET',
  url: 'http://www.someothersite.com/autos/,
  dataType: 'jsonp',
  crossDomain: true,
}).done(function(result) {
  console.log(result);
});
```

Then the URL requested will have `?callback={some $ajaxhandler}` appended

Or, you could make shorter:

```
$.getJSON("http://www.someothersite.com/autos/?callback=?", function(result) {
   console(result);
});
```

If you are implementing a web service that supports this…

- Take the specified callback (like ?`callback=mycallbackhandler`)
- Wrap the JSON you would return in a function:

```
mycallbackhandler({['data', 'some other data', 'more data']});
```

Demo

Building the Book app's UI and AJAX calls

Code can be found at:

http://coen268.peterbergstrom.com/resources/demos/booksappdemo.zip

COEN 168/268

Mobile Web Application Development

APIs and JSON

Peter Bergström (pbergstrom@scu.edu)

Santa Clara University