Full-stack Development with Node.js and React.js

IPT – Intellectual Products & Technologies
Trayan Iliev, http://www.iproduct.org/

Agenda

1. Service Oriented Architecture (SOA) & REST
2. Web Standards and Protocols (HTTP, URI, MIME, JSON)
3. REpresentational State Transfer (REST) architectural style – advantages and main constraints
4. RESTful services + JSON – lightweight and efficient way for building platform independent and loosely-coupled applications
5. Hypermedia As The Engine Of Application State (HATEOAS)
6. Sample ES6 HTTP client. Same origin policy and CORS.
7. Building simple JSON-based CRUD REST(-like) application
8. RxJS - composing functional operators. Promises
9. WebSocket
How many have implemented at least one RESTful service?
Full-stack Development with Node.js and React.js

Software Architecture – Definitions [1]

According to Roy Fielding [Architectural Styles and the Design of Network-based Software Architectures, 2000]:

A software architecture is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.

A software architecture is defined by a configuration of architectural elements - components, connectors, and data - constrained in their relationships in order to achieve a desired set of architectural properties.
Full-stack Development with Node.js and React.js

IPT – Intellectual Products & Technologies
Trayan Iliev, http://www.iproduct.org/

Slide 5

Licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

REST Architectural Properties

According to Roy Fielding [Architectural Styles and the Design of Network-based Software Architectures, 2000]:

- Performance
- Scalability
- Reliability
- Simplicity
- Extensibility
- Dynamic evolvability
- Customizability
- Configurability
- Visibility

All of them should be present in a desired Web Architecture and REST architectural style tries to preserve them by consistently applying several architectural constraints.
REST Architectural Constraints

According to Roy Fielding [Architectural Styles and the Design of Network-based Software Architectures, 2000]:

- Client-Server
- Stateless
- Uniform Interface:
  - Identification of resources
  - Manipulation of resources through representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)
- Layered System
- Code on Demand (optional)
Full-stack Development with Node.js and React.js

Classical SOA: SOAP, WSDL, UDDI

Web Services are:
- components for building distributed applications in SOA architectural style
- communicate using open protocols
- are self-descriptive and self-content
- can be searched and found using UDDI or ebXML registries (and more recent specifications – WSIL & Semantic Web Services)

Service Oriented Architecture (SOA)

Service-Oriented Architecture
A completely service-oriented model

User

Recurring ongoing cost

Services can be used and accessed through any device that hooks up to the web

Platform as a Service
E.g. Integrating with Salesforce.com's CRM

Software as a Service
E.g. Route optimizer software that uses the data to generate quickest routes

Data as a Service
E.g. Addressing data, geodata or personal data (perhaps a list of client information)

Mashups
E.g. Using Google Maps API as front-end

Author: JamesLWilliams2010, License: Creative Commons Attribution 3.0 Unported
SOA – Main Concepts

- Service Oriented Architecture (SOA)
  - Web Service
  - Web Services Registry / Repository
  - Enterprise Service Bus (ESB)
  - SOA Governance
- Contract (WSDL, WADL)
- Interface (Port Type)
- Web Service Implementation
- Business Logic
- Resources / Data

By idea from: Dirk Krafzig, Karl Banke, and Dirk Slama. Enterprise SOA. Prentice Hall, 2005

Licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Representational State Transfer (REST) [1]

- REpresentational State Transfer (REST) is an architecture for accessing distributed hypermedia web-services
- The resources are identified by URIs and are accessed and manipulated using an HTTP interface base methods (GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH)
- Information is exchanged using representations of these resources
- Lightweight alternative to SOAP+WSDL -> HTTP + Any representation format (e.g. JavaScript Object Notation – JSON)
Representational State Transfer (REST) [2]

- **Identification** of resources – URIs
- **Representation** of resources – e.g. HTML, XML, JSON, etc.
- **Manipulation** of resources through these representations
- Self-descriptive messages - Internet media type (MIME type) provides enough information to describe how to process the message. Responses also explicitly indicate their cacheability.
- **Hypermedia as the engine of application state** (aka HATEOAS)
- Application contracts are expressed as media types and [semantic] link relations (rel attribute - RFC5988, "Web Linking")

[Source: http://en.wikipedia.org/wiki/Representational_state_transfer]
## Simple Example: URLs + HTTP Methods

<table>
<thead>
<tr>
<th>Uniform Resource Locator (URL)</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection, such as <a href="http://api.example.com/comments/">http://api.example.com/comments/</a></td>
<td>List the URIs and perhaps other details of the collection's members.</td>
<td>Replace the entire collection with another collection.</td>
<td>Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.</td>
<td>Delete the entire collection.</td>
</tr>
<tr>
<td>Element, such as <a href="http://api.example.com/comments/11">http://api.example.com/comments/11</a></td>
<td>Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.</td>
<td>Replace the addressed member of the collection, or if it does not exist, create it.</td>
<td>Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.</td>
<td>Delete the addressed member of the collection.</td>
</tr>
</tbody>
</table>

Source: [https://en.wikipedia.org/wiki/Representational_state_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)
Advantages of REST

- **Scalability of component interactions** – through layering the client server-communication and enabling load-balancing, shared caching, security policy enforcement;

- **Generality of interfaces** – allowing simplicity, reliability, security and improved visibility by intermediaries, easy configuration, robustness, and greater efficiency by fully utilizing the capabilities of HTTP protocol;

- **Independent development and evolution of components**, dynamic evolvability of services, without breaking existing clients.

- **Fault tolerant, Recoverable, Secure, Loosely coupled**
Richardson's Maturity Model of Web Services


- **Level 0 – POX**: Single URI (XML-RPC, SOAP)
- **Level 1 – Resources**: Many URIs, Single Verb (URI Tunneling)
- **Level 2 – HTTP Verbs**: Many URIs, Many Verbs (CRUD – e.g. Amazon S3)
- **Level 3 – Hypermedia Links Control the Application State** = HATEOAS (Hypertext As The Engine Of Application State)

***truly RESTful Services***
Hypermedia As The Engine Of Application State (HATEOAS) – New Link Header (RFC 5988) Example

Content-Length → 1656
Content-Type → application/json
Link → <http://localhost:8080/polling/resources/polls/629>; rel="prev"; type="application/json"; title="Previous poll",
<http://localhost:8080/polling/resources/polls/632>; rel="next"; type="application/json"; title="Next poll",
<http://localhost:8080/polling/resources/polls>; rel="collection"; type="application/json"; title="Polls collection",
<http://localhost:8080/polling/resources/polls>; rel="collection up"; type="application/json"; title="Self link",
<http://localhost:8080/polling/resources/polls/630>; rel="self"
Command Query Responsibility Segregation

Considerations:


- A single representation type can not be optimal for both reading (Query) and updating the application state (Command)

- We may prefer to model Query and Command as separate resources / representation types
Web Application Description Language (WADL)

- XML-based file format providing machine-readable description of HTTP-based web application resources – typically RESTful web services
- WADL is a W3C Member Submission
  - Multiple resources
  - Inter-connections between resources
  - HTTP methods that can be applied accessing each resource
  - Expected inputs, outputs and their data-type formats
  - XML Schema data-type formats for representing the RESTful resources
- But WADL resource description is static ... let's make it dynamic!
Metadata-Representations Proposal (& Questions)

- **Meta-data can be very useful** for generic REST clients and agents crawling the Web!
- **Meta-data should be dynamically generated**
  ... but can be more stable than the data it describes
- **Separation of data and meta-data representations** (different life-cycles – allows optional retrieval, caching)
- Meta-data should be **dynamically discoverable** using hyperlinks in resource representations (rel= type/ describedby/ lrdd?)
- **Separation of Command and Query representations** -> optimal representations for each task (possibly with separate MIME types – application/vnd.*+json/xml?)

Licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
Full-stack Development with Node.js and React.js

RESTful Patterns and Best Practices


- Uniform Contract
- Content Negotiation
- Entity Endpoint
- Endpoint Redirection
- Distributed Response Caching
- Entity Linking
- Idempotent Capability
REST Antipatterns and Worst Practices

According to Jacob Kaplan-Moss [http://jacobian.org/writing/rest-worst-practices/]:

- Conflating models and resources
- Hardcoded authentication
- Resource-specific output formats
- Hardcoded output formats
- Weak HTTP method support (e.g. tunnel everything through GET/POST)
- Improper use of links
- Couple the REST API to the application
N-Tier Architectures

- Filters & Interceptors
- REST Resource Controllers
- ORM Controllers (CRUD, find/All/Range)
- MVC Controllers
- View Engines
- Entities

Client .JS

Client .JAVA

JSON/XML: HTTP/HTTPS

HTML: HTTP
Why Not HTTP - HTTP Request Structure

**GET /context/Servlet HTTP/1.1**
*Host: Client_Host_Name*
*Header2: Header2_Data*
...
*HeaderN: HeaderN_Data*

**POST /context/Servlet HTTP/1.1**
*Host: Client_Host_Name*
*Header2: Header2_Data*
...
*HeaderN: HeaderN_Data*

<Празен ред> POST_Data
Why Not HTTP - HTTP Response Structure

HTTP/1.1 200 OK
Content-Type: application/json
Header2: Header2_Data
...  
HeaderN: HeaderN_Data

```json
[
  {
    "id": 1,
    "name": "Novelties in Java EE 7 ...",
    "description": "The presentation is ...",
    "created": "2014-05-10T12:37:59",
    "modified": "2014-05-10T13:50:02",
  },
  {
    "id": 2,
    "name": "Mobile Apps with HTML5 ...",
    "description": "Building Mobile ...",
    "created": "2014-05-10T12:40:01",
    "modified": "2014-05-10T12:40:01",
  }
]
```
Cross-Origin Resource Sharing (CORS)

- Pозволява осъществяване на заявкки за ресурси към домейни различни от този за извикващия скрипт, като едновременно предоставя възможност на сървъра да прецени към кои скриптове (от кои домейни – Origin) да връща ресурса и какъв тип заявки да разрешава (GET, POST)

- За да се осъществи това, когато заявката е с HTTP метод различен от GET се прави предварителна (preflight) OPTIONS заявкка в отговор на която сървъра връща кои методи са достъпни за съответния Origin и съответния ресурс
Нови заглавни части на HTTP при реализация на CORS

- HTTP GET заявка
  GET /crossDomainResource/ HTTP/1.1
  Referer: http://sample.com/crossDomainMashup/
  Origin: http://sample.com

- HTTP GET отговор
  Access-Control-Allow-Origin: http://sample.com
  Content-Type: application/xml
Нови заглавни части на HTTP при реализация на POST заявики при CORS

- HTTP OPTIONS preflight request

  OPTIONS /crossDomainPOSTResource/ HTTP/1.1
  Origin: http://sample.com
  Access-Control-Request-Method: POST
  Access-Control-Request-Headers: MYHEADER

- HTTP response

  HTTP/1.1 200 OK
  Access-Control-Allow-Origin: http://sample.com
  Access-Control-Allow-Methods: POST, GET, OPTIONS
  Access-Control-Allow-Headers: MYHEADER
  Access-Control-Max-Age: 864000
Reactive Programming. Functional Programming

- **Reactive Programming** [Wikipedia]: a programming paradigm oriented around data flows and the propagation of change. This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow. *Ex:* \( a := b + c \)

- **Functional Programming** [Wikipedia]: a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. It is a declarative programming paradigm. Eliminating side effects can make it much easier to understand and predict the program behavior. *Ex:* `book -> book.getAuthor().fullName()`
Functional Reactive Programming: StackOverflow

According to Connal Elliot's answer in Stack Overflow (ground-breaking paper @ Conference on Functional Programming, 1997):

I'm glad you're starting by asking about a specification rather than implementation first. There are a lot of ideas floating around about what FRP is. For me it's always been two things: (a) **denotative** and (b) **temporally continuous**. Many folks drop both of these properties and identify FRP with various implementation notions, all of which are beside the point in my perspective.

"functional reactive programming" = "denotative, continuous-time programming" (DCTP)
Other Definitions of Reactive Programming

- Microsoft® opens source polyglot project **ReactiveX** (Reactive Extensions) [http://reactivex.io]:
  
  \[ \text{Rx} = \text{Observables} + \text{LINQ} + \text{Schedulers} \)

  - Supported Languages – Java: RxJava, JavaScript: RxJS, C#: Rx.NET, C#(Unity): UniRx, Scala: RxScala, Clojure: RxClojure, C++: RxCpp, Ruby: Rx.rb, Python: RxPY, Groovy: RxGroovy, JRuby: RxJRuby, Kotlin: RxKotlin, Swift: RxSwift

  - ReactiveX for platforms and frameworks: RxNetty, RxAndroid, RxCocoa

Full-stack Development with Node.js and React.js

Reactive Programming = Programming with Asynchronous Data Streams

- **Functional Reactive Programming (FRP)** [Wikipedia]: a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter). FRP has been used for programming graphical user interfaces (GUIs), robotics, and music, aiming to simplify these problems by explicitly modeling time. Example (RxJS):

```javascript
const Rx = require('rxjs');
Rx.Observable.from(['Reactive', 'Extensions', 'JavaScript'])
  .take(2).map(s => s + ' : on ' + new Date())
  .subscribe(s => console.log(s));

Result: Reactive : on Fri Jul 08 2016 14:24:25 GMT+0200
        Extensions : on Fri Jul 08 2016 14:24:26 GMT+0200
```

Good intro tutorial in RP using RxJS by Andre Staltz see: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
JS Fiddle of the demo: http://jsfiddle.net/staltz/8jFJH/48/

- **ReactiveX** is a polyglot library for composing asynchronous and event-based programs by using observable sequences.
- It extends the **observer pattern** to support sequences of data and/or events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.
- Allow composing flows and sequences of asynchronous data.
- Observables can be implemented using event loops, non-blocking I/O, actors. Client code treats all of its interactions with Observables as asynchronous, whether your underlying implementation is blocking or non.
RxJS – JS ReactiveX (Reactive Extensions)
[https://docs.google.com/document/d/1VhuXJUcILsMSP4_6pCCXBP0X5lEVTSmLivKHcUkFvFy/]

- Implementation of the functional reactive programming paradigm in JavaScript. Follows the RX (Reactive Extensions) API first introduced in the .NET framework.
- Consuming and manipulating a stream of events as collections that can be modified and transformed by various operations. A new view on the callback/promise approach.
- Follows the `OnNext()`, `OnError()`, `onCompleted()` flow.
- Aims to improve management of complex asynchronous workflows.
- Aims to solve the “callback hell” problem.
- Events don't offer the ability to be easily queried over time.
- Events are a common cause of accidental memory leaks (especially with closures).
Resources: RxMarbles and RxJS Coans

RxMarbles:  
http://rxmarbles.com/

RxJS Coans:  
https://github.com/Reactive-Extensions/RxJSKoans
Web Socket Based Communication Architecture

- Proxies: HTTP CONNECT – Tunnelling
- HTTP/S can be used WebSockets over SSL (wss:// scheme)
WebSocket Main Application Areas

- Massively multiplayer online role-playing game (MMORPG)
- Online trading – large scale auctions, stock tickers
- Interactive synchronous communication – chat, audio- & video-conferencing
- Collaborative authoring, groupware & social applications - including modelling and art
- Dynamic data monitoring and control – e.g. management dashboards presenting SLA, KPI and BI data in real time
- Remote observation and control of devices and services – e.g. remote monitoring of home security, energy consumption, data center services and devices performance visualizations
Other Alternatives for Bidirectional Communication over HTTP

- Ajax polling
- Long polling - Comet, HTTP server push, Reverse Ajax
- HTTP Streaming - Comet, chunked HTTP responses
- Bayeux protocol by Dojo Foundation – channels, publish/subscribe model
- Extensible Messaging and Presence Protocol (XMPP) over Bidirectional-streams Over Synchronous HTTP (BOSH)

Drawbacks – buffering the streamed response, HTTP request and response headers on each message, two connections – coordination overhead and increased complexity that does not scale. HTTP wasn't designed for real-time, full-duplex comm.
Full-stack Development with Node.js and React.js

Sample HTTP Request Headers – Much Overhead
IETF WebSocket protocol (RFC 6455) (1)

- Official IETF standard - RFC 6455
- TCP-based full-duplex protocol
- Starts as standard HTTP /HTTPS connection to web server port 80/443 (handshake phase) – easy firewall and proxy traversal without the overhead connected with polling
- Uses HTTP protocol upgrade mechanism (Upgrade: websocket + Connection: Upgrade) – communication is immediately upgraded to more efficient WebSocket protocol (data transfer phase)
- Allows bidirectional streaming of data (partial messages)
IETF WebSocket protocol (RFC 6455) (2)

- Designed with **security** and **extensibility** in mind: **Origin validation, sub-protocols & extensions negotiation** (through standardized HTTP headers exchanged in handshake phase)
- **WebSocket API in Web IDL** is being standardized by **W3C**
- Supported by latest versions of all major web browsers –

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hixie-75</td>
<td>February 4, 2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hixie-76</td>
<td>May 6, 2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hybi-00</td>
<td>May 23, 2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 hybi-07</td>
<td>April 22, 2011</td>
<td>6[^18]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 hybi-10</td>
<td>July 11, 2011</td>
<td>7[^19]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[^1]: Gecko-based browsers versions 6–10 implement the WebSocket object as "MozWebSocket",[^24] requiring extra code to integrate with existing WebSocket-enabled code.

WebSocket Request Example

[Request URL: ws://localhost:8080/ipt-present/ws ]:
GET /ipt-present/ws HTTP/1.1
Host: localhost:8080
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: 3RhAwlJCs7wbf3xUdeDTXA==
Sec-WebSocket-Protocol: epresentation, ipt_present
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: permessage-deflate;
client_max_window_bits, x-webkit-deflate-frame
Origin: http://localhost:8080
WebSocket Response Example

Request URL: ws://localhost:8080/ipt-present/ws

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: QIMZj0Iblv1TM+JMx/JsoSKwYb8=
Sec-WebSocket-Protocol: epresentation

Server: GlassFish Server Open Source Edition 4.0
X-Powered-By: Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.0 Java/Oracle Corporation/1.7)
W3C JavaScript WebSocket API [Web IDL]

WebSocket WebSocket(
    in DOMString url,
    in optional DOMString protocols
);

OR

WebSocket WebSocket(
    in DOMString url,
    in optional DOMString[] protocols
);
WebSocket JS API Example (1)

Example 1:
```javascript
connection = new WebSocket('ws://h2j.org/echo', ['soap', 'xmpp']);
```

Example 2:
```javascript
var rootWsUri = "ws://" + (document.location.hostname.length >0 ? document.location.hostname : "localhost") + ":" + (document.location.port.length > 0 ? document.location.port : "8080") + "/ipt-present/ws";

var websocket = new WebSocket( rootWsUri );
```
WebSocket JS API Example (2)

```javascript
websocket.onopen = function (event) {
    onOpen(event);
};
websocket.onmessage = function (event) {
    onMessage(event)
};
websocket.onerror = function (event) {
    onError(event)
};
```
WebSocket JS API Example (3)

```javascript
function onMessage(evt) {
    var jsd = JSON.parse(evt.data);
    switch(jsd.type){
    case "login-resp":
        conversationId = jsd.cid;
        $('.logged-name').html(" - " + userName);
        $('#button-login').hide();
        $('#button-logout').show();
        showToster(jsd.data.message, "info");
        break;
    ( - continues in next slide - )
    ```
WebSockets JS API Example (4)

case "logout-resp":
    conversationId = ""; userName = "";
    $('.logged-name').html('');
    $('#button-logout').hide();
    $('#button-login').show();
    showToster(jso.data.message, "info");
    break;

case "online-resp":

case "offline-resp":
    showToster(jso.data.message, "info");
    break; ( - continues in next slide - )
WebSocket JS API Example (5)

```javascript
 case "error-resp":
    showToster(jso.data.message, "error");
    break;

 console.log(jso.data);
```
References


- JavaScript Object Notation (JSON) – http://www.json.org/
Thanks for Your Attention!

Questions?