Programmable Web Project

Exercise 2

API Design

Learning outcomes (I)

- ~~Students understand what a Web API is and learn different Web API architectures.~~
- **Students understand the concept of hypermedia and how it can be used to build Web APIs.**
- **Students are able to design and implement a Web API following REST architectural style principles using existing web frameworks.**

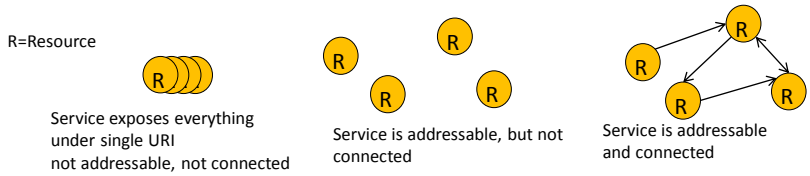
Learning outcomes (II)

- Students are able to write ~~unit~~ and functional tests to inspect their APIS.
- **Students are able to document their Web APIs using adequate software tools.**
- Students are able to implement simple software applications that make use of the APIs.

HYPERMEDIA

ROA properties

1) Addressability
2) Uniform interface
3) Statlessness
4) Connectedness



Why hypermedia?

What is hypermedia?

*"When I say hypertext, I mean the simultaneous presentation of information and controls such that **the information becomes the affordance** through which the user (or automaton) obtains choices and selects actions." - Fielding, 2008*

Hypermedia

HATEOAS?

Hypermedia as the Engine of Application State

"*A REST API should be entered with no prior knowledge beyond the initial URI and set of standardized media types […] From that point on,* ***all application state transitions must be driven by client selection of server-provided choices that are present in the received representations*** *[…] The transitions may be* ***determined by the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand)*****.**

Fielding. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Hypermedia

HATEOAS?

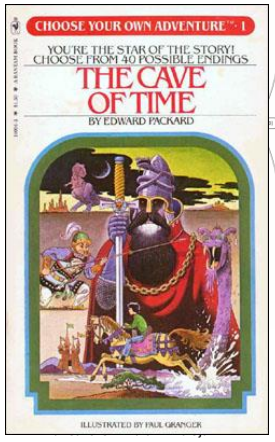
Hypermedia as the Engine of Application State

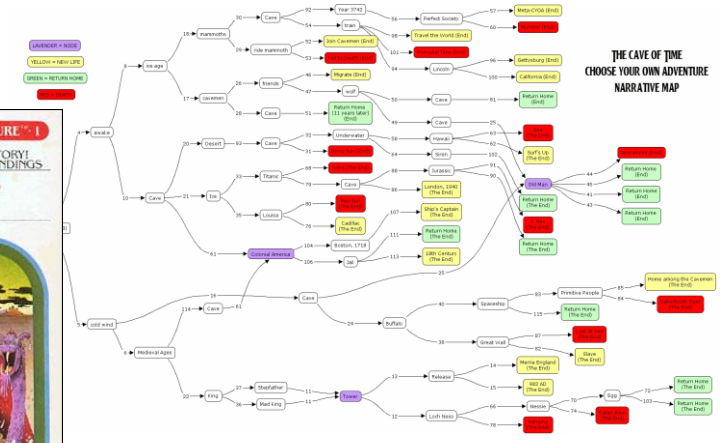
"*What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API*"

Fielding. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Hypermedia

Hypermedia As The Engine Of Application State

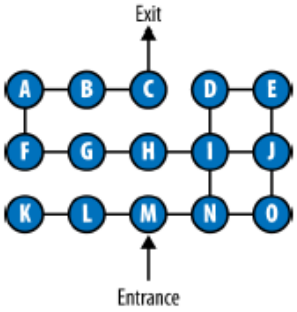




HATEOAS

- Hypermedia contains:
  - Data
  - **Hypermedia controls**
    - Links
    - Protocol specifications
- Ideally, client just need the entry point to the service
  - The rest of the URI's should be discovered through the **hypermedia controls**
    - Workflow always informed from the server
    - Server informs about possible future states via hypermedia controls
  - Well designed RESTful APIs permit **modifying the server architecture (e.g. URL structure) and data model without breaking the clients**

Hypermedia




```
<maze version="1.0">
 <cell href="/cells/M" rel="current">
  <title>The Entrance Hallway</title>
  <link rel="east" href="/cells/N"/>
  <link rel="west" href="/cells/L"/>
 </cell>
</maze>
```

Semantic Challenge !!!!

Hypermedia

When client receives a resource representation it needs to understand:

- The **structure**
  - Parse the representation
- The **application semantics**
  - Understand the representation
- The **protocol semantics**
  - Knowledge about future states
    - How can I access it: HTTP method, request format, expected response

Media type

- Defines the format of the document
  - Some types provide also protocol and application semantics
- Types:
  - General hypermedia types
  - Domain specific types
  - Patterns (e.g. collection)

Profiles

- Defines the application vocabulary and the actions I can perform in each state:
- A profile must define:
  - **Link relations**:
    - describing the state transition associated to hypermedia control (**protocol semantics**)
    - Usually implemented as '`rel`' attribute
      - http://www.iana.org/assignments/link-relations/link-relations.xhtml
  - **Semantic descriptors**:
    - Describing the meaning of properties in the representation (application semantics)

Profiles

http://tools.ietf.org/html/rfc6906

"This specification defines the 'profile' link relation type that allows resource representations to indicate that they are following one or more profiles. A profile is defined not to alter the semantics of the resource representation itself, but to allow clients to learn about additional semantics (constraints, conventions, extensions) that are associated with the resource representation, in addition to those defined by the media type and possibly other mechanisms"

Linking to a profile

- Using the **profile** Link relation:
  - RFC 6906 defines a `rel` called **profile**
  - Can be used in any `rel` attribute:
    - `links` (Siren or Collection+Json);
    - `link` defined in HTML, HAL
    - `Link` HTTP header.

```
<html>
  <head>
    <link href="http://microformats.org/wiki/hcard" rel="profile">
```

- Using the **`profile`** Media Type parameter:
  - Added as parameter in the Content-Type header

```
Content-Type = application/collection+json;profile=http://myprofile
```

- Using special purpose hypermedia controls defined in some media types.

EXAMPLES

PayPal API

https://developer.paypal.com/docs/api/overview/

HATEOAS and the PayPal REST Payment API

One of the key features of the PayPal REST API is HATEOAS (Hypertext As The Engine Of Application State).

At its core, HATEAOAS provides a way to interact with the REST Payment API entirely through hyperlinks. With each call that you make to the API, we'll return an array of links that allow you to request more information about a call and to further interact with the API. You no longer need to hard code the logic necessary to use the PayPal REST API.

HATEOAS links are contextual, so you only get the information that is relative to a specific request.

Building Blocks of HATEOAS

Let's take a look at a PayPal payment using the REST API. When you send the initial payment request, you'll get back a response that contains a set of HATEOAS links like this:


```
[
  {
    "href":"https://api.sandbox.paypal.com/v1/payments/payment/PAY-6RV70583SB702805EKEYSZ6Y",
    "rel":"self",
    "method":"GET"
  },
  {
    "href":"https://www.sandbox.paypal.com/webscr?cmd=_express-checkout&token=EC-60U79048BN7719609",
    "rel":"approval_url",
    "method":"REDIRECT"
  },
  {
    "href":"https://api.sandbox.paypal.com/v1/payments/payment/PAY-6RV70583SB702805EKEYSZ6Y/execute",
    "rel":"execute",
    "method":"POST"
  }
]
```

PayPal API

https://developer.paypal.com/docs/api/overview/

rel component

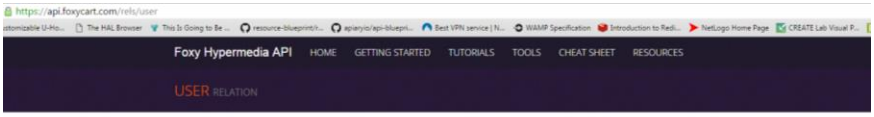
The `rel` component provides the relation type for the URL in question.

Here are the possible relation types:

- `self`: Link to get information about the call itself. For example, the `self` link in response to a PayPal account payment provides you with more information about the payment resource itself. Similarly, the `self` link in the response to a refund will provide you with information about the refund that just completed.
- `parent_payment`: Link to get information about the originally created payment resource. All payment related calls (`/payments/`) through the PayPal REST payment API, including refunds, authorized payments, and captured payments involve a parent payment resource.
- `sale`: Link to get information about a completed sale.
- `update`: Link to execute and complete user-approved PayPal payments.
- `authorization`: Link to look up the original authorized payment for a captured payment.
- `reauthorize`: Link to reauthorize a previously authorized PayPal payment.
- `capture`: Link to capture authorized but uncaptured payments.
- `void`: Link to void an authorized payment.
- `refund`: Link to refund a completed sale.
- `delete`: Link to delete a credit card from the vault.

FoxyCart

https://api.foxycart.com/docs/resources



Description

To modify a user, you must be connected to the API using an OAuth token with user_full_access scope.

A FoxyCart User is someone who manages and configures a FoxyCart Store. It is often the owner of that store, but might also be someone paid to be responsible for aspects of the store. There can be many users per store and any user can access many stores. Before creating a user, you must create a client. When you create a user, you'll get a response back containing an OAuth token giving you access to the user.

Sandbox Example

You can interact with this resource and run actions against the sandbox API via our HAL Brower.

Actions

GET	View a user
PATCH	Update a user (send only the properties you want to modify)
PUT	Replace a user (send the entire representation)
DELETE	Delete a user
HEAD	Get just the header response
OPTIONS	Get a response explaining which HTTP methods are supported

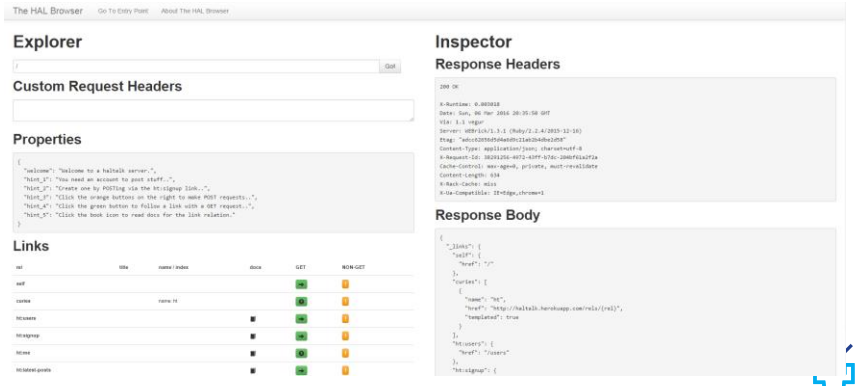
Properties

Property	Description	Type	Constraints
first_name	The user's given name.	String	Required. 50 characters or less.
last_name	The user's surname.	String	Required. 50 characters or

HAL Browser

https://github.com/mikekelly/hal-browser

- http://haltalk.herokuapp.com/explorer/browser.html#/
- https://api-sandbox.foxycart.com/hal-browser/browser.html#/



HOW CAN I DESIGN AN API USING HYPERMEDIA?

How can I design an API using hypermedia?

1. Find a media type that best suits my needs
   – If I cannot find it create your own specific domain design
2. Find a profile that best suits my needs
   – A profile must define:
     • Protocol semantics (if the media type does not provides it).
       – Usually using rel relations
     • Application semantics
       – Define the semantics descriptors
   – If there is no a good profile from me:
     • I can create a new profile from a existing ones. I need to clarify the subset of the existing profile I am using.
     • I can create a new profile from several existing ones. I need to clarify the subset of the existing profiles I am using
     • I can create a new profile from scratch.

How can I design an API using hypermedia?

Whatever you do be sure that you provide

1. The format of the representation (in the media type)
2. The protocol semantics (either in the media type or in the profile)
3. The application semantics (either in the media type or in the profile)

What type to choose

http://sookocheff.com/post/api/on-choosing-a-hypermedia-format/

API DOCUMENTATION

WHY DOCUMENTING WEB APIS USING SPECIFIC DOCUMENTATION FRAMEWORKS.

Swagger

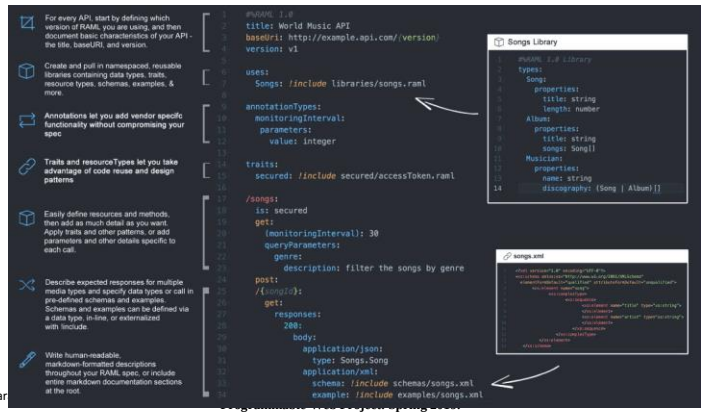
http://swagger.io/

- Complete framework implementation for describing, producing, consuming, and visualizing RESTful APIs.
- **Documentation embedded directly in their source code**, preventing out-of-sync between documentation and code.
- Includes set of tools such as Swagger UI to visualize APIs, Swagger editor to design an specification from Scratch using **YAML** and Swagger Codegen to generate client/server implementations.

RAML

http://raml.org/homepage

- **YAML** based language for describing RESTful APIS
- Include tools such as API console or HTML generator



API Blueprint (I)

- https://apiblueprint.org/

- Documentation-oriented web API description language based on markdown language.
- API blueprint decouple elements of API to enable modularity while encapsulating backend implementation behavior.
- API Blueprint provides tools for the whole API lifecycle. It can be used to discuss your API with others, generate documentation automatically, or a test suite.

API Blueprint (II)

Specification: https://apiblueprint.org/documentation/specification.html
Tutorial: https://apiblueprint.org/documentation/tutorial.html

IMPORTANT NOTATION:

- **Resource Groups:** List a set of related resources:

```
# Group Questions
Resources related to questions in the API.
```

- **Resources:** Individual resource information. It is associated to a unique URI / URI template. Uri parameters are explained in the Parameters section

```
## Question information [/questions/{question_id}]
      + Parameters
        + question_id (number) - ID of the Question in the form of an integer
```


API Blueprint (III)

- **Action:** Defines a complete HTTP transaction as performed with the parent resource section. **An action must include at least one response.** It **may include multiple responses, with different status codes.**

```
### List All Questions [GET]
```

- **Request:** HTTP request-message example payload

```
### Add question [POST]
You may create your own question using this action.
+ question (string) - The question
+ choices (array[string]) - A collection of choices.
    + Request (application/json)

            { "question": "Favourite programming language?",
              "choices": [ "Swift",”Ruby”] }
```

API Blueprint (IV)

- **Response:** HTTP response-message example payload

```
### Add Question [POST]

   + Response 201 (application/json)

       + Headers

               Location: /questions/1

       + Body

           { "question": "Favourite programming language?",
             "choices": [ "Swift","Ruby"] }
```


APIARY

https://apiary.io/

- Online API design stack that enables collaboration in different phases of API design and implementation
  - It uses API Blueprint and Swagger
- Multiple tools to enhance API generation:
  - API Editor
  - Mock Server
  - Apiary CLI
  - API inspector
  - Interactive documentation
  - Automated Testing
  - Integrated code examples