Reykjavík University
T-514-VEFT Web Services
Assignment: Small Assignment I
Due: 13th of September, 2018

HYPERMEDIA-DRIVEN RESTFUL API DESIGN PROCEDURE FOR 'BUGBUSTER'

Good fundamental design for web services can be vital for their performance, reliability and their maintainability. In this assignment the methodology for designing a RESTful Web API introduced in the 9th chapter of the book *RESTful Web APIs: Services for a Changing World* by authors Leonard Richardson, Mike Amundsen and Sam Ruby is exploited in a step-by-step manner. The exploitation focuses on the web service design procedure, meaning that step seven of the methodology is omitted as no implementation nor code is involved, but a thorough description of how each step of the design process would be carried out if the example provided for a web service would be implemented by this methodology

Alexandra Geirsdóttir
alexandrag16@ru.is

Edda Steinunn Rúnarsdóttir
eddasr@ru.is

Smári N. Guðmundsson
smaring16@ru.is

PART ONE

Bugbuster Web Service

To exploit the design process methodology we provide a step-by-step design process for a web service that enables a system called **Bugbuster**. Bugbuster is a JIRA-like productivity tool where users working on bug fixes can keep track of the state of their bug fixing tasks. Bugbuster consists of a work board divided in three sections, **"backlog"**, **"working"** and **"done"**, to which a user can assign 'sticky notes' representing some tasks.

STEP 1 Semantic Descriptors

First things first we identify the resource components the Bugbuster web service will contain and their hierarchical structure. We assume that the Bugbuster will hold multiple work boards, so an entry point to Bugbuster is most likely a list of **work boards**. Each work board will contain a list of (exactly three) **sections**. Each section needs to somehow be differentiated from other sections and receive an order as 'backlog' (ordered first), 'working' (ordered second) and 'done' (ordered third and last) section via some property in each section - let's call this property **description** for the time being. Sections will contain a list of **tasks** as well (i.e. the 'Sticky Notes'). What type of section a task is in defines the task status. Each of these tasks will have identifying property **title**, a property **description** and a property **assignee** (a property which is not required and can be omitted and/or undefined). This forms an easily distinguishable hierarchy, see diagram 1.1.

DIAGRAM 1.1
Semantic Hierarchy

- WORK BOARDS list of all work boards
  - WORK BOARD a single work board with sections
    - SECTIONS list of sections in a work board
      - DESCRIPTION identifies section as the backlog/working/done section
      - TASKS list of section's 'Sticky Notes'
        - TITLE task title
        - DESCRIPTION task description
        - ASSIGNEE task assignee (omitted if unassigned)

STEP 2 State Diagram

If we look at the semantic descriptors from step one, we see that the hierarchy is very distinguished and can help us figure out the potential flow of our hypermedia-driven API which we can visualize via State Diagram. The most logical entry point to API would be the top component: list of all work boards from which, given a hypermedia-driven flow, next step would be to retrieve a single work board.

Once a single work board is retrieved, it's logical to assume that further information on that board is retrieved; the list of sections on the board (the backlog/working/done sections, in that order) where each section contains a list of tasks for these sections.

Then the last logical step of the flow is retrieving a single task from a section's list of tasks. Now according to our semantic hierarchy we've arrived at our innermost component. For each task there exist two actions: the **assign** action which changes a task's assignee and the **change status** action which moves a task from it's section ('backlog', 'working' or 'done) to another to update it's status (note that this action will change assignee to unassigned if task is moved from section 'working' or 'done' to 'backlog' as a side effect). Both these actions are the only ones in our system that are not read-only, but unsafe and idempotent.

From these steps we construct a state diagram; see diagram 1.2.

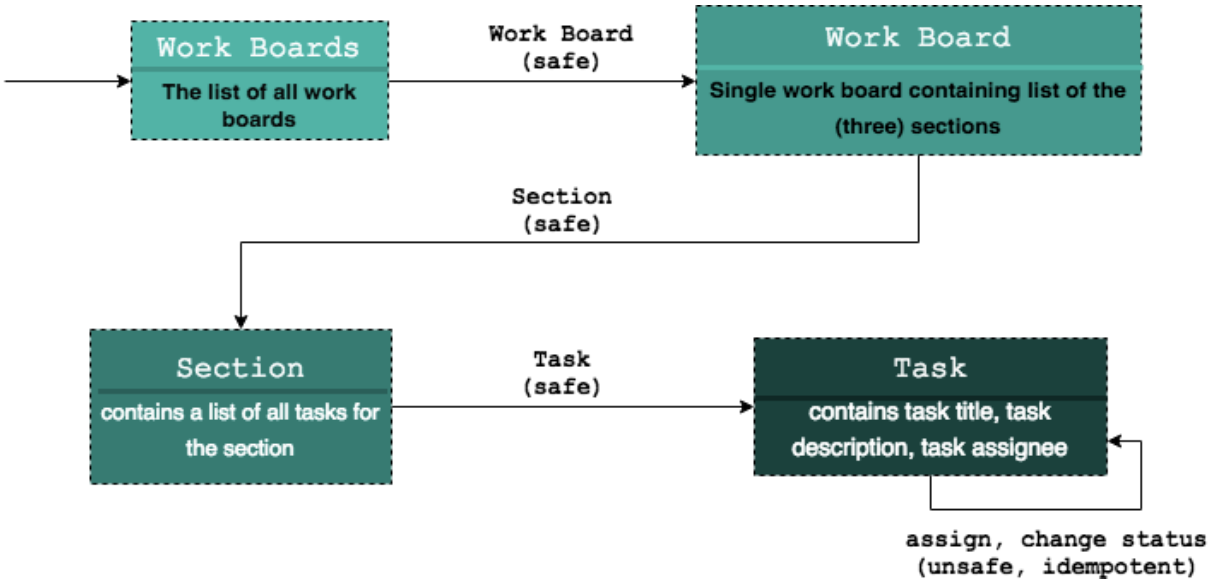

DIAGRAM 1.2 State Diagram for Bugbuster Web Service

STEP 3 Name Reconciliation

The names assigned to our API's resources in steps one and two are now reviewed with the intention to make them more human-readable or straightforward to users.

Names given to API's components in steps one and two are quite straightforward for describing their purposes and quite human-readable. Actions in the state diagram (such as *re-assign* and *change section* actions on tasks) also fairly indicate whether they are safe or idempotent via their name which is why we leave actions' names unchanged in this step.

However, there is one component that can be named in a more descriptive way: the description property of a section. The name is misleading in consideration to human-readability - it's easy to think that this property is a sentence describing a section whereas it's intended to identify a section as 'backlog', 'working' or 'done' with some order. We therefore rename the '**description**' property of a section to '**phase**' as a more straightforward name for the property indicating that this is a value indicating progress towards some goal (here the 'goal' or highest phase would be the 'done' phase), i.e. it is a phase with some name and some order. We adjust our semantic description according to the name reconciliation procedure as shown in diagram 1.3.

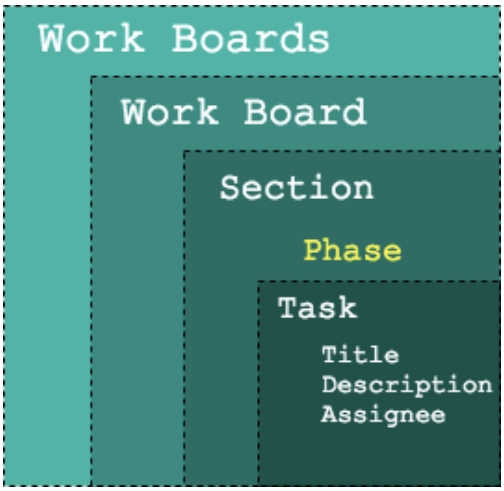

DIAGRAM 1.3 Updated Semantic Hierarchy

STEP 4 Choosing a Media Type

Various hyperlink media types can be used for web services, with some of the main ones being JSON-LD, Collection+JSON, SIREN and HAL which we will now assess and choose the one we find most appropriate for the design of our API.

Our API design has two unsafe idempotent actions: *reassign* and *change status* of a task. This makes JSON-LD hyperlink media type unfit for our web service due to the fact that this media type cannot represent unsafe state transitions/actions on its own. So although we could add an additional system such as HYDRA to add support for actions for JSON-LD, we attempt to find a less complex solution for the sake of simplicity and a more directly fitting media type.

Collection+JSON supports editing and deleting items as well as adding items to collections which is what our web service needs to support. Also, our state diagram implements a what could be a collection pattern. This makes Collection+JSON fit for our web service. However, given our hierarchical structure the application semantics would probably be rough, not very human-readable and too technical with this media type. Using Collection+JSON therefore might turn out like a raw interface into the web service database making it hard on the API users.

We now explore HAL, the leading hypermedia type. It's lightweight and enables resources to have a state and list of links that lead user to additional resources and embedded/children resources of the current resource. It's therefore easily understandable which is good for human-readability. There's one drawback to HAL however: there's not enough support for actions. That is quite bad for our web service since we have two said idempotent actions that we'd like users to be aware of. We therefore rule out HAL as our media type. SIREN on the other hand is similar to HAL, it leads users on to embedded/child resources via links and is just as easily understandable for the human user, but tries to eloquently fix HAL's drawback by adding support for actions and introduces the classes to a JSON model, bringing a sense of type information to responses. It therefore would fit our web service.

CONCLUSION We are left with Collection+JSON and SIREN. Despite that both could work for our web service we arrive at the conclusion of using **SIREN** as the media type. It's easier to use, easier for human users to fully grasp and understand and with it we can define all actions we need to define our web service. Collection+JSON would have been an appropriate choice as well, but in our biased opinion we find human-readability in APIs too valuable to forfeit.

STEP 5 Writing a Profile

Since no additional application semantics in our web service design are present, apart from those additional semantics that are SIREN specific of which profiles exist on, writing a profile for the web service is not appropriate in this case. SIREN even has the power to provide a very clear, human-readable documentation via the action property it provides on actions that can be made on API and therefore on state transitions shown in the diagram.

As the SIREN profile explains, the hypermedia type adds the properties "**links**" to reference resource and/or embedded/children resources, and "**actions**" to define which actions can be made on the resource.

Although there is no point in repeating existing profiling of SIREN semantics, we provide an example on it's semantics for the sake of clarification. The following SIREN JSON object could be an example on how our resource **Task** may be sent as response to a user. It has additional property "links" with hypermedia driven links to itself and embedded resources and "actions" property defining actions (an example of how the *assign* action could be formatted is shown in example).

```
/**** EXAMPLE MODEL OF A HYPOTHETICAL 'TASK' RESOURCE FROM WEB SERVICE ****/
{
  id: 1
  title: "example task"
  … /* MORE RESOURCE PROPERTIES */
  links: [
      {
        rel: [ "self" ],
        href: "https://api.bugbuster.com/workboards/1/sections/1/tasks/1"
      },
      … /* MORE LINKS */
  ],
  actions: [
      {
        name: "assign",
        title: "Assign a bug to someone",
        method: "PATCH",
        href: "https://api.bugbuster.com/workboards/1/sections/1/tasks/1/assignee",
        fields: [
          { title: "bugTaskTitle", type: "text" },
          … /* MORE FIELDS */
        ]
      }
       … /* MORE ACTIONS */
  ],
}
```

Moreover, we want to explicitly notify API users that the resources retrieved use SIREN as hypermedia type so that the client will know how to properly parse and/or utilize the resources the web service retrieves. Therefore the following header is added to the web service's responses:

```
Content-Type: application/vnd.siren+json
```

(***source***: documentation from official developers of SIREN, see here: https://github.com/kevinswiber/siren)

This will explicitly notify the client that it is dealing with a SIREN hyperlink format data when resources are retrieved, meaning all additional semantics to a resource such as action and links should be clear to the client.

STEP 6 Implementation

As this assignment focuses on the design procedure of a web service first and foremost, implementation will not be provided for the Bugbuster web service.

However if we were to implement a web service for Bugbuster it would be beneficial to use the ***ExpressJS*** web framework which comfortably works alongside NodeJS to get the job done. To make the code extra sexy TypeScript would be used for the server instead of JavaScript so that it's possible to make TypeScript interfaces as to make the code more type-safe.

This is due to ExpressJS being a minimalist and lightweight framework that makes writing web services quite easy for developers with minimal effort to set up the framework and to develop such a server. ExpressJS servers also yield minimal overhead for requests and responses and since our web service does not have relatively high complexity anyway this is a desirable factor as well.

STEP 7 Publication

BUGBUSTER BILLBOARD URL https://api.bugbuster.com

The Bugbuster web service should now be ready in this phase of the design procedure so we publish it by providing an URL as an entry point into it. We choose this URL in terms of readability and simplicity, https://api.bugbuster.com**,** indicating that it is a web service and is for the Bugbuster tool. As shown in the state diagram for the service (see: diagram 1.2), the entry point into our web service is to the list of all work boards in the system.

Logically, sending requests to this URL gives users insight into how to further navigate the API as it is hypermedia-driven so this URL would act as a hypermedia gateway into our web service. To provide an example on how this works, sending a GET request to the gateway URL would logically return a list of *all work boards*, that would provide links further into the service i.e. how to retrieve a *specific work board (*f.x. via link *https://api.bugbuster.com/workboards/{id})*, which would provide links to embedded resources such as that *work board's sections (*f.x. via link *https://api.bugbuster.com/workboards/{id}/sections)* and so on and so forth. Therefore it would be redundant to publish more URLs as they are explicit.

DOCUMENTATION Swagger Documentation

It can be time-consuming for developers to write their own web service documentation which is why various tools are available to help generate such documentation almost automatically, that can make the API's actions readable and understandable just as well. *Swagger* is a leading open-source API documentation generation tool in the modern world, and a powerful one because it generates documentation readable both by computers and humans. The tool can provide clear and well-defined documentation on the capabilities and actions of a web service without granting any direct access to implementation (i.e. source code, network access, documentation).

Since our goal is to reduce the amount of time to accurately document a service and to minimize the amount of work needed to connect disassociated services we choose ***Swagger*** to generate documentation of the actions and capabilities of our Bugbuster web service and we include the Swagger documentation in the publication of our service.

HYPERMEDIA FORMAT SIREN

No new hypermedia type was created for the purposes of designing this web service. If one had been created, the new media type would be appropriately registered with the IANA and a thorough documentation on the new media type would be included along with the Bugbuster web service publication.

But since we chose an existing hypermedia format SIREN for our web service, as described thoroughly in step four, this is not needed. However, we include in the documentation of the web service that it uses the hypermedia format of SIREN. We also include links to existing profiling/documentations on how SIREN works so users know how SIREN augments our web service. Note that publishing our hypermedia format is rather a formality or informational addition to the documentation of our web service but not really a necessity. This is because the users retrieving resources should already be aware that SIREN is used in our API since we use the SIREN Content-Type header (as stated in step five):

```
Content-Type: application/vnd.siren+json
```

PROFILING (Not Appropriate For This Design Procedure)

As stated before in step five, no specific profiling was deemed necessary for the Bugbuster web service and so no profiling is included in the publication of this design procedure.

If specific profiling had been a part of the procedure it would be published along with the web service to demonstrate profiling-defined link relations and semantic descriptors and registered online (if appropriate) for reusability.

PART TWO

Expanding Bugbuster

Now that the concept for the design process for Bugbuster has been introduced, we want to explore the procedure when the Bugbuster tool is expanded. The Bugbuster wants to introduce Q & A section for work boards. In order for the web service to function with the new feature the hypermedia-driven flow is now re-evaluated.

CHANGES TO HYPERMEDIA-DRIVEN FLOW (None Required)

The result of our re-evaluation of the hypermedia-driven flow and design is that given how we have designed the system in part one **no changes are required** for the hypermedia-driven flow nor the design procedure if new Q&A section is added to the system. The one thing that potentially changes in the system by this addition is the server's **resources**, but that is not related to the hypermedia-driven flow nor to the API design itself so that is irrelevant to this re-evaluation.

To further explain, no changes are required because adding a new section to the system with explicit ordering (i.e. before 'done' and after 'working' sections) will only require adding a new section-resource to our work board resources with a new type of property **phase**. This is because in the way we've designed the system in part one of this assignment, the phase property of the sections provides explicit ordering of sections and despite the new section being added, the older phases ('backlog', 'working' and 'done' phases) will remain and a new type of the phase property with new ordering will exist ('Q&A'). The system will still work exactly how we've describe that it would, except that of course now when users request sections of a work board they will receive a list of four sections with four different phases instead of three sections .

Therefore our design concept for the Bugbuster API remains as it is when this expansion is made to the Bugbuster tool.