Web Platform API Design Principles and Service Contract

Yonghong Chen, Xiwei Xu, Liming Zhu
NICTA, Australia Technology Park, Eveleigh, Australia
School of Computer Science and Engineering, University of New South Wales, Australia

Abstract—The emergence of web services gives rise to service ecosystems, where core service platforms are exposed through APIs to a large number of third parties to provide new solutions. This often requires platform APIs to be able to evolve rapidly while not breaking backward compatibility in order to support a wider range of planned and unplanned use. The REST architecture style allows a service contract to be negotiated and delivered at runtime aiming to achieve higher level of evolvability. However this potential is often not fully realized due to the lack of systematic approach in architecture design to separate requirement-related contract elements from specification-related elements and integrate them into the overall API design. In this paper, we present a novel approach to designing service platform APIs in order to achieve high evolvability. We firstly apply the problem frames approach to better understand the interface coupling and the evolution of requirements. Based on the problem frame analysis, we propose a media-type centric approach to designing RESTful service API. We applied our approach to the Twitter API evolution. The result shows significant improvements in the areas of contract leaking prevention, interface consistency and adding new features.

Keywords—REST; Platform API; Evolvability; Service contract

I. INTRODUCTION

The emergence of web services gives rise to service ecosystems [1], where core service platforms are exposed through APIs to external parties and their communities of users to provide solutions for market niches in order to respond to increasing competition and economy pressure [2]. The service platform and its environment evolve in concert. The platform APIs have to work with third party services and support new business requirements especially in a very dynamic, highly diversified and uncontrolled market environment. The aim of evolvability is to enable continuous quality and functionality improvement while controlling the complexity at the same time [3]. This means the platform architecture and its API need to allow both internal rapid development and occasional external API function improvement while not significantly breaking existing third party applications already using the APIs.

There are two different classes of web services used in service platform APIs identified by W3C [4]: 1) REST-compliance web services, the primary of which is to manipulate web resources via a uniformed set of “stateless” operations; 2) arbitrary web services, which may expose arbitrary set of operations. In the arbitrary web services, the interoperability of machine-to-machine interaction is typically based on interface description languages such as WSDL. By mutually agreeing on a set of rigidly defined service interfaces, the internal implementation on both client and server sides can evolve independently. However, the interface itself is brittle to change. In contrast, REST is aimed to reduce design time contract information to a small set of core business specific knowledge. Detailed contract information is only composed from pre-established contract knowledge and delivered at run time by embedding dynamic contract information in the payload itself [5].

Existing approaches to designing RESTful platform APIs [6, 7] are typically divided into the following steps: 1) resource abstraction and identification; 2) resource representation and media types design; 3) HATEOAS design [8] to connect resources and convey application states. These steps provide flexible ways of expressing business concepts most syntactically in its API. However these steps do not answer a more fundamental question of what business concepts and to what level of details are required to be understood by the clients at the semantic level of the API. And the sequence of steps is also problematic when media-type and HATEOAS design is considered very late in the design [6]. Both are the common causes of contract leaking issue. For example, in Twitter, a tweet is identified by a numeric number as unique id. Initially, developers were not clear how many bits a tweet id is defined and assumed it was 32 bits. Later, twitter changed the way the id was generated in 2010 and settled on 64 bits due to its underlying database system and performance trade-offs [7]. But it caused problems with Javascript clients which failed to parse numbers more than 53 bits. To avoid such problem, one important question should be considered during the API design: is it necessary for clients to understand the semantic concept of tweet id to consume the services?

Another issue of designing platform API is the interface inconsistency. Maintaining a consistent interface for similar functionality across different modules improves API usability and reduces client’s development effort. However, different non-functional requirements often cause different API designs for the same functionality, which often result in inconsistent interfaces. In Twitter platform, there are in total six different parameters used to implement paging facilities across different modules (See
Table 6 for more details). For example, “rpp” is used in searching API, while “since_id” and “max_id” are used together in timeline APIs to provide a more efficient way (non-functional requirements) of browsing highly dynamic data without repeating old data. The API design of the shared functionality often involves trade-offs of other qualities, such as usability or efficiency, to promote generality. A more fundamental problem here is to improve API consistency and reusability while supporting extensions to satisfy non-functional requirements and minor functional variations.

The third issue of designing platform API is the adding of new features in an open environment. For example, Twitter platform originally only support posting short messages within 140 characters. Many third party services are developed to support pictures, videos and, even online polling surveys. These services greatly enriched existing features and became very popular. In response to market pressure, Twitter implemented native support for embedding media entities in text messages. However, it only supports a small subset of media formats. In addition, it did not provide a compatible solution for third parties to extend the range of currently supported media formats. The fundamental problem here is how to design the API to allow diverse feature extensions from uncontrolled, unplanned, and independent development.

In this paper, we present a novel approach of platform API design to improve evolvability and demonstrate its capabilities by addressing the above three issues. This approach contains three steps.

Firstly, we use problem frames [9] to decompose the problem into layers of problem domains. Requirements and specifications (all relative to each other depending on the level of abstraction) are decomposed progressing gradually from high-level business requirements down to low-level programming constructs and specification. The design focus also progresses gradually from external requirements down to specific internal implementation details. Then output is a tree of problem domains with semantic concepts called “phenomena” as domain interfaces. Feature dependencies, requirement couplings and contract leaking are fundamentally due to cross referencing and interaction of phenomena among domain interfaces [26]. This layering process enables designers to better understand the coupling among domains and avoids accidental contract leak. Backward compatibility can be also significantly improved due to reduced coupling.

Secondly, a set of media types are produced as service contract. The data schema and processing models are defined based on semantic concepts extracted from domain interfaces. We also propose “visibility control” as a novel runtime API extension mechanism to achieve forward compatibility. This mechanism takes advantages of the problem domain tree produced from the previous step. By setting different visibility levels for different API elements at runtime, implementation details for different levels of problem domains can be hidden from each other even within the same API.

The third step is the traditional REST step on resource abstraction and identification. We don’t claim any contribution on this. However, our approach relies on problem frames and a media-type-centric design to provide inputs to the resource abstraction and identification step.

This paper is organized as follows: Section 2 gives the background knowledge. Section 3 discusses the detail of the new design approach. In section 4, we evaluate the proposed approach by retrospectively redesigning the Twitter platform APIs for evolvability and comparing it with the real Twitter APIs in terms of addressing historical requirements changes and evolution needs. Section 5 surveys several related works. Section 6 concludes the paper.

II. BACKGROUND

A. Problem Frames

Problem frames is an approach to software requirement analysis. A problem is located in the world outside the machine to be built while the solution is implemented within the machine [9]. The boundary between the machine and the external world is explained using the problem progressive diagram in Fig.1. The machine on the bottom represents what we build in software, the requirement of which refers just to the machine and completely ignores all problem domains. Any problem is somewhere on a progression towards the machine. The top problem is the deepest into the problem world, the requirement of which refers to Domain A. By analyzing the Requirement A and the Domain A, Requirement B (next domain towards the machine) can be found that refers only to Domain B and guarantees satisfaction of the Requirement A. Eventually is the machine at the bottom. Problem progression explains the dilemma [10] between what and how. “what something does” as oppose to “how it does it” is entirely a matter of perspective [11]. We use problem frames to decouple different concerns in our approach.

![Figure 1. Problem Progression](image)

The interfaces between the machine and problem domains are described as phenomena, which are something observed to happen in a time point or existing in a period of time. The core of phenomena is the distinction between requirements, specifications, and
domain properties [12]. There are two categories of phenomena. The first category is individual, which is something that can be named and reliably distinguished from other individuals. There are three kinds of individuals, namely, events, entities and values. The second category is relation, which is a set of associations among individuals. This category also consists of three different sub types, namely, states, truths and roles [11]. We use these concepts to capture different API semantics in our approach.

B. Platform API Design

A center principle of platform API design is to separate API and service contract from actual implementation and prevent implementation details from leaking into the API [13]. The platform API should only reflect the external requirements as it describes “what” is required and provided by a platform. At the same time, it should be agnostic to the specification which describes “how” the platform implements and satisfies the requirements. Although some implementation details, such as private data structure/operations and language/systems choices, are easy to identify and not leaked, optimizing for non-functional requirements through APIs and subtle grey areas between requirements and specification at different levels of abstraction are much more difficult to identify and prevent from leaking and coupling. We gave some motivating examples in the Twitter case in section I.

Problem frames and phenomena based understanding of requirements provides a systematic approach to two things. First, it helps properly structure the requirements and separate the requirements from specifications relatively along the progression of each level of problem domains. Thus, developers are able to better encapsulate service components into APIs at different levels to achieve higher level of reusability and substitutability. Second, it helps better understand the root cause of coupling and dependencies, rather than treating dependencies and constraints as black box entities [12]. Shared phenomena among problem domains serve as basic units of domain interfaces, which enable developers to detect dependencies and plan ahead at early design phases to provide forward compatibility for future evolution.

However, even if all requirements are properly framed and separated according to the problem progression in layers, APIs might still be polluted by implementation details. One example is the implementation detail introduced by conversational context. For example, if one requirement is about message creation and a second requirement is about message reading, commonly, an identifier would be introduced into the APIs as conversational context so that creating and reading operations can both refer to the same message. A second example is the implementation detail introduced by trying to optimize for non-functional requirements through the API. For example, if the message arriving rate is very high, the system may wish the identifier to incorporate creation time for easy caching. From a client’s perspective, the existence of these identifiers is implementation detail and not of its own concern. However, such identifiers often appear in the API unavoidably. But from the platform’s perspective, it is desirable to enforce restrictions to prevent clients from relying on the semantics of the identifiers.

As suggested in [12], feature dependencies are fundamentally due to cross referencing same phenomena or related phenomena from different requirements at different layers/domains. These dependencies are often reflected in the APIs and pollute one layer with another layer’s relative implementation details. A mechanism is needed to formally separate implementation dependent and independent elements within an API, and communicate the information to clients. Clients should be restricted from interpreting implementation dependent elements of an API while still enabled to correctly invoke services with correct parameters. This would prevent accidental service contract leak especially across different layers in the API.

C. REST

REST is an architecture style, which promotes no prior knowledge of the services beyond the initial URL and a set of standardized media types that are appropriate for intended audience [5]. REST can separate implementation from API by encapsulating resources behind resource presentations and preventing typed resources or fixed resource names. In REST, implementation dependent API elements can be visible to a client, but REST requires the client to ignore them and only process the resource presentations according to the media types. Hypermedia enables a client to correctly invoke services without the necessity to interpret implementation dependent API elements. This is fundamentally different from a SOAP-based Web service API, in which every parameter and data structure must be understood by the client. Thus, REST is much more evolvable due to better isolation of implementation details.

However, there are two challenges to truly achieving the evolvability potentials of REST. Firstly, REST does not provide any guideline of how to differentiate implementation dependent implementation details from requirements. Secondly, media type is a static document shared within a digital community, while the boundary between implementation independent requirements and implementation dependent specifications is purely a perspective matter. The requirements of one domain could be the specifications of another, especially in an open platform community, where services often consume other services to provide integrated solution to clients. Thus, to enable independent third party extensions, the perspectives of different domains need to be taken into considerations. A mechanism is needed to control the visibility of implementation dependent API elements according to the domain perspective and selectively enable them to be modified or replaced at runtime.
III. ARCHITECTURE DESIGN APPROACH

Our architecture design approach is divided into three steps. Firstly, we use problem frames to analyze the requirement following the divide and conquer strategy. The problem models produced in the first step are used in the second and third steps to design media types and identify resources separately. The generated media types are regarded as the service contract, which is required to be communicated to the consumers to collaborate with the service platform.

A. Problem Analysis

We first propose a new problem frame for the problem analysis to specifically address the synchronous interaction pattern of REST. Since REST does not support server data push, the clients in a distributed environment have to actively initiate polling to retrieve updates of domain properties in order to make observations on service domain. Both functional and non-functional requirements may be imposed on the system to define what and how data should be polled and displayed to users effectively and efficiently.

The new composed problem frame combines a “work pieces” problem frame and an “information display” problem frame [9], which enables separating the concern of how information is displayed and what is displayed, as shown in Fig.2. Within the “work pieces” problem frame, a set of computer processable objects can be edited. This problem frame enables editing the configuration of how and when information should be displayed. The “information display” problem frame only addresses the issue of what information is to be rendered and presented to external world. In REST, a resource may have multiple representations and can be negotiated at runtime depending on client capability. Thus, our “information display” domain does not address the rendering data format. We only examine what domain properties the clients would be interested from requirement perspectives.

![Figure 2. Composite Problem Frame](image)

For example, a business process engine manages a set of processes, tasks and cases. We can apply this composed problem frame to enable a client to observe current status. The requirements we could analyze include a) non-functional requirements such as how often domain properties can change and properties updates can be polled; b) functional requirements, which are the domain properties to be presented to the client, such as current executing tasks, elapsed time, estimated time and etc.

We follow the approach in [14] to recursively decompose requirements from an abstract business level to concrete system requirements following the problem progression model. Firstly, at the outmost layer, we decompose the system using abstract business goals. Each sub problem has a goal which answers the question of what is to be achieved. Secondly, we identify specific requirements applicable to each sub problem domain. Based on these requirements, shared phenomena among sub problem domains and external world are extracted and documented as output. Thirdly, implications and design assumptions for each domain are explicitly identified. Requirements are transformed into specifications based on the design assumptions [14]. This is to define how the goal can be achieved and decomposed into sub problem domains. Fourthly, the specification of the parent problem domain becomes the requirements of the child problem domains. The process repeats until the transformed requirement references only phenomena at the interface of the machine for all problem domains. As the output of the entire process, we obtained a problem domain tree and shared phenomena of each problem domain.

Our approach emphasizes the distributed nature of service platforms and introduced the composed problem frame to address the synchronous nature of REST interactions. We also identify shared phenomena and use them in the media type design step along with the problem domain tree.

B. Media Type Design

The second step of the design approach is media type design, which is on the basis of the problem models generated in the previous step, and is aimed to generate the exact service contract. From the previous step, we have collected a set of phenomena as the interfaces of domain interactions. This set of phenomena is the minimal set of vocabularies that the media types need to support. Thus, clients can interact with the system without any out-of-band information outside of the media types, which satisfies message self-descriptiveness constraint of REST.

Designing a media type involves two parts, namely, mapping extracted phenomena to media type data schemas, and designing processing models of data schemas which means defining what methods are applicable for the resources of that media type. In REST, the available methods for a resource must be specified by hyperlinks unless they can be mapped to HTTP verbs.

The media type design starts from the outer most problem domain and progress down toward the machine side. For each level of the progression, we map phenomena to data schema to determine how information is rendered to external clients. Entity phenomena [9] are mapped as container elements of the schemas, while states are mapped as data fields or attributes of parent entities. Values and truths are phenomena which do not change over time. They often define default values or value sets of states. To design the processing models of data schemas, events and roles are mapped to links that define what methods can be applied to a data type. To improve interoperability [15, 16], it is desirable to map value, state, event phenomena into standard vocabularies defined in
public standards or ontologies as much as possible. For example, vocabularies from xhtml can be used for color, font value phenomena and vocabularies from APP [17] can be used for pegaing events such as “first”, “next” and “last”. The whole process repeats until all levels of problem domains are completed. Multiple media types may be produced as the end result. However media type vocabularies must be partitioned in accordance to the partition of the problem domains.

Listing 1. Visibility Control

At runtime, we introduce visibility attributes to entities in media types to allow “relative” implementation details to appear but control their visibility to clients. It also allows the platform provider and different third parties to have more fine-grained control over the API and external solutions. This is very similar to access controls in object-oriented programming. Listing 1 gives an overview of the visibility control for an entity. An entity may have four optional attributes:

- Visibility – Possible values for visibility are: private, protected and public. The default value is public when it is not specified. The content of private entity is purely internal and shall only be interpreted and used by the original owner, which is specified in “owner” attribute if present. The content of a public entity can be parsed by clients or any immediate REST component according to media type specification. The content of protected entity can only be directly parsed by the REST component that is capable of processing the media type specified in “type” attribute. Both the client and the server may inject protected entities into the resource representation. Any REST component that is not capable of parsing the protected entity should simply ignore it and passed it on when applicable.

- Type – the media type of the entity content.

- ProcessEngine – A URL that specifies the address of a default process engine of the entity content.

- Owner – An URI identification string that uniquely identifies the owner of the entity content.

Consider a very simple media type describing a message as shown in Listing 2. The media type is used in a digital community with the following parties: a) The platform frontend that provides an overall solution; b) The platform backend message storage service, which is controlled by the same business entity as the platform and simply provides storage service and sort the message by time for better caching/paging efficiency; c) The third party extension service, which provides a customized extension to support embedded smiley; d) The clients that simply use the service to send, reply and display a message.

Listing 2. Message media type

The visibility attributes inside the media type make the different perspectives of the various parties clear. For example, the clients and the third parties have no interest of what id the message has or when it is created. Thus, MessageId is marked as “private” with the owner “www.platform.com” specified, which allows both front end and back end of the platform to interpret it. The “CreateTime” is also marked as “private”, however, with the owner of “www.platform.com/backend”, which prevents the frontend from interpreting it. From the perspective of the frontend, how messages are efficiently cached and paged is implementation detail.

The platform utilizes the third party to extend the feature without knowing the detail. The frontend and backend of the platform together provide solutions in different problem domains at different levels of problem progression models. When client sends a message using the third party extension service, the third party registers itself in the protected element “EmbeddedMedia” and implements the smiley extension. Both the platform frontend and the client can retrieve a rendered representation of the message via invoking the service at www.example.com and sending the entire EmbeddedMedia content as parameter.

C. Resource Abstraction and Identification

Resource identification is considered as implementation in our design process and beyond the scope of this paper. As highlighted in [5], server must have full control of the namespace of resources. A client should discover the resources by interacting with the service rather than relying on fixed resource names or hierarchy. We refer readers to [6, 7] for general guidelines of resources identification.

D. Discussion

Our design approach is a contract first service platform design approach, while existing approaches are usually experience-based and often explore evolvability from syntactic level, such as designing flexibility data schema and injecting version numbers. Our approach explores evolvability from semantic level and focuses on aligning evolvability with business objectives. By using problem progression models, domain interfaces are identified systematically and gradually gravitate down to syntactic programming constructs. It separates requirements from
specifications at different layers and enhances reusability and extensibility at each level of problem progression.

Existing media types do not have visibility control. In fact, the platform, third parties and clients all have different perspectives for the interaction. The client interacts with service to complete business operation with no interest of implementation details. Third parties could provide solutions for a particular feature extension, hence is only interested in a particular portion of the media types. Even for the platform, different internal development teams in charge of modules correspondent to different problem domains in the problem progress model may have different perspectives. In addition, the same media type could be reused in a completely different context. Our approach recognizes the differences in perspective and allows the contract to be explicitly controlled with visibility attributes at runtime. This reduces coupling and significantly improves overall evolvability and reusability.

IV. EVALUATION

A. Twitter platform

To illustrate our approach and evaluate its effectiveness, we retrospectively redesign the REST API of Twitter. Twitter is a popular open platform which offers microblogging service that enables users to post short text based messages up to 140 characters, known as “tweets”. Message content is delivered to potentially interested users by being organized into timelines. Twitter is claimed to have 660k developers and 900k applications. In addition it is consumed by thousands of third party web services and mash-ups, according to ProgrammableWeb [18]. Any changes to the API could have wide impact and long term ripple effect to many external applications and web services. Despite the effort of preventative measurements and planning, changes are often forced to rollback due to the extreme difficulty of managing such ultra large size digital community outside its enterprise boundary. In this evaluation, we first present out approach and improved API, then, we illustrate how it can minimize the chance of breaking existing clients and improve evolvability.

B. Problem Analysis

We assume the distinguishing business strategy of Twitter platform is to discover and share what is happening and actively push content in real time to potentially interested users rather than passively waiting for users to search for contents. Problem domain of Twitter can be roughly divided into two sub-problems. The first sub-problem is tweet editing that involves creating and editing tweet messages. The second sub-problem is how to deliver contents to potentially interested users. In Twitter this is done by organizing tweets in timelines. For the purpose of evaluation, we extracted requirements from existing twitter API and manuals through reverse engineering.

1) Tweet editing sub-problem:

Listing 3 gives the requirements related to tweet editing. The tweet editing sub-problem is a typical “work pieces” problem frame, where two domains can be identified as shown in Fig.3, namely, client and the display. The client issues commands to edit tweets, while the display shows if the command is successful. The extracted phenomena are shown in Table 1. Typically, the nouns in the requirements can be extracted as entity or value phenomena such as tweet, user and message. Verbs are often good candidates for events. Conditions and contextual statements can be captured as states and roles.

Listing 3. Requirements for tweet editing

R1.1: A logged in user can perform actions to create, update and delete a tweet
R1.2: A logged in user can retweet, reply an existing tweet.
R1.3: Tweet Messages must be shorter than 140 characters.
R1.4: Appropriate error message will give given to user when operation fails. For example, message is over-length or user is not logged in.

Figure 3. Problem model for tweet editing

Table 1. Phenomena for tweet editing

<table>
<thead>
<tr>
<th>Entity</th>
<th>Tweet, user</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>LoggedIn(user), Original(Tweet), OperationSuccessful(operation), Message(string), ErrorMessage(string),</td>
</tr>
<tr>
<td>Value</td>
<td>MaxMessageLength(140),</td>
</tr>
<tr>
<td>Event</td>
<td>Create, Delete, Update, Retweet, Reply</td>
</tr>
</tbody>
</table>

2) Timeline sub-problem: Timeline is a collection of tweets sorted by time in descending order. Timeline service can be decomposed recursively using our composite problem frame. In such composite problem frame, real world state and behaviour information is continuously needed and presented via the display sub-domain, in which structured data is offered as service observable to external world. The configuration editor sub-domain addresses domain interfaces involved in setting up complex configurations for polling and displaying data records structurally and efficiently. For example, in order to provide an easy interface to allow one user to subscribe to messages posted by a group of other users, functional features, such as social graph and user groups, are introduced into the system. Paging feature is also introduced into the system to enable browsing data records. Non-functional requirements are subsequently imposed on paging features to only fetch data during paging to avoid redundant data transmission and improve overall system efficiency.
The configuration editor sub-domain for timeline is a complex domain which requires further decomposition. There are roughly four different categories of specifying user’s interest and delivering message contents accordingly, as shown in Fig.4: a) user-based – if a user is interested in another user or a group of other users, he may be interested in his or their messages; b) content-based – a user may be interested in a message if it contains topic or elements which match his expressed interests; c) trend-based – a user may be interested in popular or most recent messages; and d) interaction-based – a user may be interested in messages that are interacted with him, such as favorite or reply actions.

![Figure 4. Problem decomposition of timeline](image)

After framing the configuration editing of timeline, we repeat the same process to identify phenomena from requirements for each sub-problem domains. However, we want to bring attention to the paging feature which appears in all the display domains in Fig.4. Paging related phenomena across each individual display domain are identical. For example, the display timeline domain displays a collection of tweets which are highly dynamic data with very high incoming rate. Thus, non-functional requirement is imposed to improve system efficiency and scalability. However, displaying users list in a user group does not impose such efficiency requirement as user list updates infrequently. Listing 4 gives the paging relevant requirement. The phenomena extracted from the requirements are shown in Table 2. The non-functional requirement would be anticipated to be realized in cursor implementation and value initialization.

![Listing 4. Requirements for paging](image)

| R2.1: Timelines must provide paging facility to allow user to retrieve new data from current position without repeating old data. |
| R2.2: User group can optionally provide paging facility for better usability. |
| R2.3: Users can specify number of records to display and browse forward and backward. |
| R2.4: Default page size is 20 for timeline and 5000 for users list. |

**C. Media type design**

From problem analysis, we have identified the minimal set of phenomena as domain interfaces. In the media type design, it is relatively straightforward to convert them into vocabularies for data schema and processing model.

1) **Data Schema Design**: Display domains are the starting points for data schema design. In these domains, we analysed what information is to be presented in the previous step. In this step, we start with collecting entity and state phenomena and structuring them as data schema that are to be used in resource representations. A subset of entity and state phenomena from tweet and timeline display domains are as listed in Table 3.

![Table 3. Entity and state phenomena from display domains](image)

<table>
<thead>
<tr>
<th>Entity</th>
<th>Timeline, Tweet, User</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>LoggedIn(user), Original(Tweet), OperationSuccessful(operation), Message(string), ErrorMessage(string)</td>
</tr>
</tbody>
</table>

2) **Processing model**: Events and roles are natural candidates for link relationships, which identify what methods are applicable to a resource. A subset relating to tweet and timelines are shown in Table 4.

![Table 4. event phenomena](image)

<table>
<thead>
<tr>
<th>Paging</th>
<th>Forward, Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tweet Messaging</td>
<td>Create, Reply, Retweet, Update, Delete</td>
</tr>
<tr>
<td>Timeline Events</td>
<td>Favorite</td>
</tr>
</tbody>
</table>

3) **Value and Constants**: A subset of value phenomena is shown as inTable 5. They often appear as default values or possible values for states.

![Table 5. value phenomena](image)

<table>
<thead>
<tr>
<th>Timelines</th>
<th>PublicTimeline, UserTimeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rendering</td>
<td>Colors, Fonts</td>
</tr>
<tr>
<td>Tweet</td>
<td>MaxMessageLength</td>
</tr>
<tr>
<td>Paging</td>
<td>DefaultPageSize</td>
</tr>
</tbody>
</table>

4) **Select standards** – For better interoperability, we select the following standards for vocabulary reuse [5]:

- Xhtml for font, color and other rendering related lexical phenomena because most of our clients are web browsers.
- APP for paging and browsing data records. Phenomena such as “Forward” and “Backward” and mapped to APP vocabularies “previous” and “next”.
- Create/Update/Delete are mapped to standard HTTP verbs
- Operation status and error messages are mapped to standard HTTP status codes and messages.
D. Resource Identification

Since display problem domains identified what information clients would be interested to see, these domains are also good places to start performing resource abstraction and identification. Entity phenomena identified in these domains are natural candidates for resources, such as timeline, tweet, user, user lists and etc. The URL assignments are purely an internal matter as clients must not be coupled to the resource URLs or hierarchies.

E. Case Study

1) Case Studies 1 – Prevent contract leak: In this simple example, the tweet element has a default visibility level of public. But tweet “id” is marked as private to prevent other software engines from parsing or using it. Links are provided to “retweet”, “reply” or “favorite” current tweet. We did not provide links to “create”, “update” and “delete” because they are standard HTTP operations. The allowed methods could be inquired by sending OPTIONS request.

2) Case Studies 2 – API consistency

Listing 5. An representation of timeline resource

```
<Timeline>
  <Category>=""PublicTimeline""</Category>
  <Pagination>
    <limit>20</limit>
    <cursor visibility=private>
      <since_id>1000</since_id>
    </cursor>
    <Link rel="prev" url="/PublicTimeLines?page=4&since_id=1000" rel_templates="/PublicTimeLines?page={page}&since_id={since_id}">
    </Link>
    <Link rel="next" url="/PublicTimeLines?page=6&since_id=1000" rel_templates="/PublicTimeLines?page={page}&since_id={since_id}">
    </Link>
    </Pagination>
  </Timeline>
```

Listing 5 gives a representation of timeline resource using our approach. The “limit” is a public element which controls the number of data records displayed per page. The cursor is marked as private. This forces the visibility level of all its child elements, “since_id” and “page” in this case as private. How many variables to use and the choice of them are implementation details. This gives the platform the flexibility to choose different paging implementations to satisfy different non-functional requirements without introducing extra coupling. This also informs the client explicitly not to rely on the current cursor API implementation. The client should simply follows the links to browse forward or backward. The client can also change number of data records per page by using the url template and the limit variable.

3) Case Studies 3 – Design for extension: Previously, we mentioned many third party services extending text based message service to provide embedded media. We provide a sample solution to allow such extensions to be integrated with native embedded media support. In existing Twitter platform, a tweet may include a number of media entities. A native media entity can be an URL link, a username, a trend topic or a media file such as jpeg. A user can upload a tweet with an embedded jpeg through the API “/statuses/update_with_media”.

Listing 6. An representation of tweet resource

```
<tweet>
  <message> Hello </message>
</tweet>
```

As shown in Listing 6, we implemented a tweet representation which contains two media entities. The first entity embeds a JPEG in the tweet, the second entity embeds a third party polling extension. Both entities have common public properties such as offset, displayUrl and etc. These common properties allow a client to relate embedded media entities to the text message. The differences between these media entities are with the content elements which are marked as protected. The first content element has a type of “image/jpeg”, indicating only a process engine capable of rendering jpeg image may parse the data contained within the content element. The content element in the second media entity example is a 3rd party polling entity. It requires a process engine that is capable of parsing a vendor specific media type “application/vnd.polling”. It also suggests a default rendering engine at “http://example.com/pollingService”.

The application can send the entire content to this rendering engine in order to retrieve a rendered output. However, the client should not attempt to direct parse the content element. With such API design, we allow native platform support of media type to coexist with third party extensions in a consistent way. The extensibility is greatly improved.

F. Discussion

1) Enforce the constraints of REST: Compared to original Twitter API, where many information are eliminated from out-of-bound documentation such as: available timelines, supported interactions, available parameters and rate limits, Message self-descriptiveness and HATEOAS constraints are naturally enforced by our approach. Combined with normalized value objects, HTTP verbs can be easily mapped to perform CRUD type of operations. Thus, full conformance of REST uniformed
interface constraints is enforced via the architecture method systematically.

2) Avoid contract leaking: Twitter ID is an implementation detail which can be marked as private to prevent client from parsing the value. Thus, we systematically avoid implementation contract leak as per final API implementation.

Table 6. Six parameters of paging in existing Twitter API

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of records to retrieve</td>
</tr>
<tr>
<td>Since_id</td>
<td>Return results greater than specified id</td>
</tr>
<tr>
<td>Max_id</td>
<td>Return results less than specified id</td>
</tr>
<tr>
<td>page</td>
<td>The page of the result to retrieve</td>
</tr>
<tr>
<td>rpp</td>
<td>Return per page. Used in search API only</td>
</tr>
<tr>
<td>Cursor</td>
<td>Only applicable when retrieving user list. It breaks results into pages of 20 users per page for subscribe API, 5000 users for friends API.</td>
</tr>
</tbody>
</table>

3) Interface consistency: In the original Twitter platform, six parameters are specified in its APIs as shown in Table 6. Twitter developer forum also documented three different paging mechanisms just by combining the first four parameters, since it is not clear what are valid combinations of these parameters. Besides, the paging interfaces are inconsistent across different features. We demonstrated how to produce the consistent paging API by marking implementation specific elements as private.

4) Extension of new features: Feature extensions from third parties can be enabled in a compatible way by defining protected API elements and allowing third parties to override the contents and register correspondent media types and processing engines.

V. RELATED WORK

ROA (Resource Oriented Architecture) [6] has wide spread impact on both industry and research communities. ROA takes advantage of resource addressability as the primary method of achieving evolvability. Strategies and plans for versioning control and forward compatibility such as inserting version numbers into URL is usually advised to be considered in early design stages [6]. Initial ROA approach [6] does not provide guidelines of deriving resources from function specification [19] and plays down the importance of message self-descriptiveness and HATEOAS constraints. It ignores the requirements of pre-established common vocabulary in message self-descriptiveness constraint and the reuse of link relationship in HATEOAS constraint. The design process of resource identification and service contract is experience-based and does not provide a mechanism for contract extension. Furthermore, the delivery mechanism of service contract heavily relies on out-of-band information. As result, the evolvability potential of REST is severely undermined.

To improve the resource abstraction design process in ROA, many approaches are proposed to derive resources from functional specifications from both data-centric perspective [20, 21, 22] and control-centric perspective [23, 24, 25]. In data-centric resource abstraction approaches, supported resource types are often predefined in isolation of business requirement. Therefore, it is difficult to effectively and efficiently model resources to fulfill business requirements. Furthermore, there is lack of mechanism to extend supported resource types for further evolution. While in control-centric approaches, definitions of resource types are ad-hoc and diversified. Hence, it is difficult to support message self-descriptiveness constraints due to lack of common vocabulary.

In a HATEOAS driven approach, however, the emphasis of web service contract is on reusable contract-centered media types [26]. By standardizing and creating reusable contracts, composition scalability [15, 27] can be improved significantly. Service contract design in HATEOAS-driven approaches focuses on answering “how” to discover available services by interacting with it and following hyperlinks. This enables design time service contract to be limited to a set of pre-established media types and common vocabularies to serve as the syntactic and semantic foundation of the ecosystems around the web platform to maximize its evolvability [5]. This pre-established contextual contract information allows further dynamic contracts to be composed, extended and discovered by the clients at runtime. Recent REST research papers also turned their focuses on hypermedia-driven approaches to design highly decoupled and flexible web services [28, 29, 30]. However, most of them only considered message self-descriptiveness constraint at syntactic level while ignoring self-descriptiveness and reusability at semantic level.

Problem frames approach has been applied in the arbitrary web services to align service implementation with the business objectives [31] and describe the capability of available web service [32]. However, to the best of our knowledge, problem frames have not been used in the design of REST-compliance web services.

VI. CONCLUSION AND FUTURE WORK

In service ecosystems, the platform APIs work with third party services and support new business requirements in a very dynamic and uncontrolled market environment. Thus, the platform API is required to allow both internal rapid development and occasional external API function improvement. RESTful web service, as one of the possible implementations of service platform API, can reduce design time contract information to a small set of core business specific knowledge and compose and deliver detailed contract information at run time. A common criticism of REST is that the business requirement and logics often cannot be specified sufficiently by shared media types. In our view, business requirements can be reusable if the implementation is abstracted away.
In this paper, we use problem frames approach to systematically identify shared phenomena in the business requirement and abstract away solution specific requirements. The identified phenomena through problem analysis enable designers to better understand couplings and dependencies among domains at very early design stage. Our visibility control provides a mechanism of clearly defining different perspectives of service contract for different components during the interaction.

We evaluated our approach through retrospectively redesigning the REST API of Twitter Platform. The result shows that our design approach can enforce the uniformed interface constraints of REST and significantly improve evolvability in terms of preventing contract leaking, keeping interface consistency and enabling new feature extensions.

In our future work, we plan to investigate the feasibility of using ontology to document phenomena and their interactions to enable designer to visualize the dependency. This would be able to provide a feedback loop to evaluate and improve requirement engineering process.

ACKNOWLEDGMENT

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

REFERENCE