MockAPI: An Agile Approach Supporting API-first Web Application Development

José Matías Rivero[1], Sebastian Heil[2], Julián Grigera[1],
Martin Gaedke[2], and Gustavo Rossi[1]

[1] LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
{mrivero,julian.grigera,gustavo}@lifia.info.unlp.edu.ar
[2] Department of Computer Science, Chemnitz University of Technology, Germany
{sebastian.heil,martin.gaedke}@informatik.tu-chemnitz.de

Abstract. In the last years, agile development methodologies have been widely adopted. However, they still lack support for API requirements while, at the same time, public RESTful APIs are fueling a rapid growth of web applications providing services built on other services. On the other hand, whereas Model-Driven Development techniques successfully increase the productivity in the development of data-intensive web applications, they lack the agility required when developing heterogeneous web applications with frequent requirement changes. In this paper we introduce MockAPI, an approach based on annotating user interface mockups that combines the advantages of agile approaches and Model-Driven Development. We introduce a metamodel for annotations and demonstrate how to derive running API prototypes as starting point for agile development. RESTful API best practices and API-first development are introduced into the agile process. The MockAPI approach defines a set of constraints to accelerate the development of web applications. We also show the results of a brief validation applying MockAPI to popular web sites.

Keywords: API, Model-Driven Development, Agile Development, Prototyping.

1 Introduction

Agile development methodologies have shown a massive adoption [1] because they allow to adapt quickly to changing requirements, effectively shorten the development cycle and include end-users more intensively in the development process, in order to reduce risks during projects. However, these development approaches are lacking support for API-related requirements (i.e. stating what the applications should provide as a service and how), since their advantages are not efficiently applied when gathering and implementing requirements that are not strictly related to user interaction (like user interface or business logic), i.e. not related to *what the user can see* [2].

Infrastructure-as-a-Service (IaaS) and Software-as-a-Service (SaaS) are transforming the way of providing services in the Cloud, and at the same time dropping the costs. On the one hand, IaaS provides a fast, easy and cost effective way of requesting infrastructure (processing, storage, data transfer, etc.) as needed to implement and

F. Daniel, P. Dolog, and Q. Li (Eds.): ICWE 2013, LNCS 7977, pp. 7–21, 2013.

scale applications. On the other hand, SaaS provides working software over the Cloud at a low cost, avoiding higher cost of deployment and maintenance associated with custom on-site installations. In this way, IaaS is providing an important cost reduction for software developers while SaaS is providing a similar reduction for software end-users. As a consequence, since both trends are intended to avoid an on-site installation of infrastructure and software, they provide APIs to facilitate critical operations like servers instantiations, storage increment requests (IaaS), data exportation/importation or special data operations (SaaS) [3].

On the application level, APIs are commonly used for different purposes. A common API layer is usually built to fulfill the business requirements of applications that run on different platforms and front-ends (web, desktop, mobile, etc.). Besides, making APIs publicly available is a well-known way of extending the impact and use of popular applications. Main examples of this approach are Facebook through its Facebook Graph API[1], and Twitter[2].

The development of APIs is getting more attention because they speed up the development process allowing reusing already existing software and infrastructural power to deliver software faster through integration of existing components. As the API becomes more important from the strategy and technology point of view and is part of the requirements, the challenge is to either help the end-users understand the concept and hidden complexity of distributed systems, or to provide a way for retrieving the necessary information for the API design with common requirement gathering techniques. Current agile methodologies do not provide a way of gathering and structuring this kind of requirements. Agile methodologies leave all the APIs definition work for developers without any guidance. Since in API-First[3] development the implementation of a core API is a blocking task delaying other tasks like frontend development, the entire development process is slowed down.

In this paper we provide a structured way of dealing with the definition of APIs, from requirements gathering to implementation. In addition to textual user stories, we use annotated user interface sketches (mockups) of the different front-ends of the application. We do so in order to gather a general overview of the underlying API of the application being built. The annotations placed over mockups can be easily applied to textual user stories as well, working as a story *stereotyping* strategy.

The rest of the paper is organized as follows: in Section 2 we analyze related work and background of the fundamental concepts used in our proposal, Section 3 details the core features of our approach including procedural and technical features. In Section 4 we explain implementation details and Section 5 summarizes the results of a validation experiment featuring popular real-world web sites. Finally, in Section 6 we conclude the paper and envision future work.

---

[1] Facebook Graph API - `https://developers.facebook.com/docs/reference/api/`, last accessed on 06-March-2013.

[2] Twitter Developers - `https://dev.twitter.com`, last accessed on 23-Feb-2013.

[3] API-First development - `http://www.api-first.com`

2 Background and Related Work

2.1 State of the Art in Web Applications Development

When developing software through direct coding, extensive tool support and well-known practices are often available to make the development process faster and less error prone for developers. Dependency management tools, Integrated Development Environments (IDEs), build and deployment environments among many other remarkable tools are available to assist the development team daily. In the same sense, a plethora of technologies, patterns, practices and processes have been defined to cope with complexities in software development like Design Patterns, Aspect-Oriented Programming, Test-Driven Development, etc. However, while they substantially help developers in the process, there are still many challenges related to coding software by hand: writing it syntactically and semantically correct according to the elicited requirements, writing tests to check whether the application meets them, use the same patterns, practices, programming style and frameworks in the correct way, etc. To make things worse, integration between newly developed software and SaaS applications (from social networking like Facebook or Twitter to infrastructural services as provided by Amazon[4], Microsoft[5] or Google[6]) are becoming increasingly required in industry. This introduces the problem of interacting with other software using particular communication channels and data formats.

Not specifically focused on APIs, Model-Driven Development (MDD) [4] solutions have been defined to cope with such challenges. In MDD, software is defined as a set of high-level models and derived automatically using code generators, respecting a previously agreed architecture, patterns and platform defined by the software architects or developers. The main problem in MDD is that it only allows specifying software features by concepts included in the high-level language. When a special feature has to be included in the application, either the language has to be extended in order to express and further derive this features or the generated code has to be modified manually. MDD can be suitable for specific types of development such as data-intensive web applications [5]. However, MDD is less applicable for developing heterogeneous applications. This is due to the cost of personalizing the MDD infrastructure to cope with detailed and rapidly changing requirements and implementations. In a previous work we explore the possibility of bringing an agile approach to MDD [6], starting from mockups to gather requirements and generating prototypes.

On the other hand, software scaffolding solutions like Ruby on Rails[7] propose an intermediate solution: they allow generating the structural parts of the applications expressed in some simplistic specification language (sometimes using standards like XML or YAML). Once generated, they have to be manually refined by developers,

[4] Amazon Web Services - `http://aws.amazon.com/`, last accessed 23-Feb-2013.

[5] Windows Azure - `http://www.windowsazure.com/en-us/`, last accessed 23-Feb-2013.

[6] Google App Engine - `https://developers.google.com/appengine/`, last accessed 23-Feb-2013.

[7] Ruby on Rails - `http://rubyonrails.org/`, accessed on 28-Feb-2013.

discarding the initial specifications. Such approaches force a specific platform and architecture with the advantages of automatic code generation to speed-up the initial stages of the development. Similar to scaffolding approaches, user interface prototype annotations like Canonical Abstract Prototypes [7] intend to model common UI patterns and propose a semi-automatic code generation. However, since they focus on user interface implementation – that is inherently complex – they only allow generating a limited subset of features, leaving the task of dealing with the generated UI code to the programmer.

An additional issue in all three approaches is the need to manually translate requirements (expressed usually as user stories, use cases, natural language narratives, etc.) to code or models only observing the requirements artifacts; that is, no assistance is provided to guide this process. In this work, we present a Model-Driven process that allows defining and quickly generating an initial API for a Web Application to speed-up the initial iterations in the development. This allows developing the application frontend that uses the API through direct coding speedily in order to obtain a fully functional running version of the application that can be tested with end-users as soon as possible. Finally, the generated API can be further partially or totally implemented as necessary in the following iterations. Thus, our approach intends to combine classic code-based development with Model-Driven and Scaffolding processes.

2.2 Agile Development Style Meets Service-Oriented Architecture

In agile development, the focus on a rapid implementation of functionality that yields a visible business value can be unfavorable in the context of service-oriented architecture [8]. While user stories are customer-oriented, architectural aspects like identification and modeling of services, data resources or API design are not covered in agile development [9–11]. Though there are proposals to tackle service related features in the early requirements gathering stage, e.g., using use cases [12], they do not use requirement artifacts fully understandable for end-users, which are at the same time, unambiguous and technically sound for developers [13]. Approaches like [11] advocate using architectural knowledge bases for decision making and evaluation, however, they do not focus on accelerating the process to create early running versions.

Advantages of the API-first paradigm cannot be fully leveraged. Ideally, common functionality and resources for different application platforms are consolidated at the service layer. This enables independent parallel development of applications for different devices and facilitates serendipity through the development of third-party applications benefiting from the exposed service layer [14].

As API development requires a lot of experience and knowledge about best practices [15], API quality in agile development is highly dependent on the developer team's skill level. There is no process-intrinsic guidance or widely accepted concept available that supports agile developers in using best practices.

Agile development teams encounter difficulties when applying a service-oriented architecture style. Particularly, there is a gap between requirements represented by customer-oriented stories and application architecture, which can produce poorly designed APIs. Application of best practices for a clean, usable API is highly dependent on the experience of the development team as there is no further guidance

provided. For better support of applying service-oriented architecture style in agile development, a refined approach is required bridging the gap between requirements and architecture by combining the most promising elements of various development approaches employed today and providing enhanced guidance regarding API best practices to the agile developer.

3 The MockAPI Approach

In this section we describe motivation, procedural and technical aspects of our approach that allows quickly specifying and generating APIs using requirement artifacts that are easily understood by both developers and customers: user interface mockups.

3.1 The Approach in a Nutshell

To overcome the issues mentioned in Section 2, we proposed an approach called *MockAPI*. MockAPI aims at helping developers in an agile environment to design service-oriented applications. The proposed process starts by eliciting requirements through user stories and their related user interface mockups. Such mockups represent an intermediate language between developers and customers, being technically sound to developers and fully understandable by customers [13]. Mockups are then annotated with simple but formal specifications that we use to automatically generate a first API implementation. This API is intended to help building the first iterations of the different application front-ends, reducing the requirements-to-software time and effort, though it might be later replaced by the definitive one. In this paper we focus on generating APIs for service-backed web applications, however, the same annotation approach can be used to generate other artifacts like interaction descriptions related to mockups (that can be checked by end-users) or data layer schemas and configurations.

3.2 MockAPI Process

To exemplify the approach within an agile methodology we chose Scrum, since it is one of the most widely adopted in industry [1]. The Scrum process starts with the construction of a Product Backlog, listing Stories, ordered by value delivered to the customer. Then, the product is built iteratively in Sprints. Every Sprint starts with a Planning Meeting in which Stories are selected from the Product Backlog according to their priority and broken down into Tasks, forming the Sprint Backlog. A short Daily Meeting is held every work day to gain awareness of work progress/problems. At the end of each Sprint, a potentially shippable application is demonstrated to the Product Owner and customer [16].

The MockAPI Scrum process in Fig. 1 proposes using mockups in all steps. Since a mockup represents the user interface/interaction required to satisfy a story, mockups form an intermediate tool between abstract stories and concrete tasks. Therefore, we propose to add mockups to the Sprint Backlog. Mockups must be built and annotated with stakeholder participation; developers can explain semantics if needed.

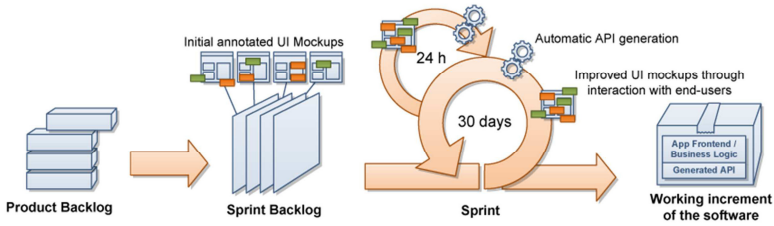

Fig. 1. An overview of the MockAPI Scrum process

The developer team starts with coding the application front-end. An initial API implementation can be derived from annotated mockups to speed-up the process. Thus, in early iterations, the development team can focus on interaction and presentation allowing for early feedback. Front-ends for different devices (e.g. cellphones, tablets, PCs) can be built in parallel with API support from the outset.

Although changes in mockups are frequent, they do not require strong re-implementation effort: the API can be re-generated from updated annotations.

3.3 Mockup Building and Annotation

MockAPI relies on annotating mockups to discover and specify features related to the required API. Annotations can serve both as requirements and implementation specification. In the following subsections we describe the structure of the annotations MockAPI defines to specify API-related features.

3.3.1 Dealing with Content

One of the basic specifications required to define an API is its content (in terms of types and relationships) and the way it is accessed. To deal with these concerns, MockAPI proposes the following annotation types, depicted in Fig. 2 over sample mockups for a conference management system:

List(ItemName): describes a list of items in a mockup, of the type *ItemName*. For instance, the `List(conference)` tag in the leftmost mockup from Fig. 2 denotes a list of `conference` objects. From these tags, we can infer the existence of resources called *ItemName* (objects of type `ItemName`) aggregated in a list.

Item(ElementName): expresses that the annotated mockup shows a user interface containing representation of a single item called *ElementName*. A mockup showing the details of a conference is annotated with `Item(conference)`.

Viewing/Editing: describe access type to resources; we identified two basic resource access patterns: `viewing` and `editing`. Both are included as tags in MockAPI. `viewing` represents read-only access, `editing` represents Create, Read, Update, Delete functionality.

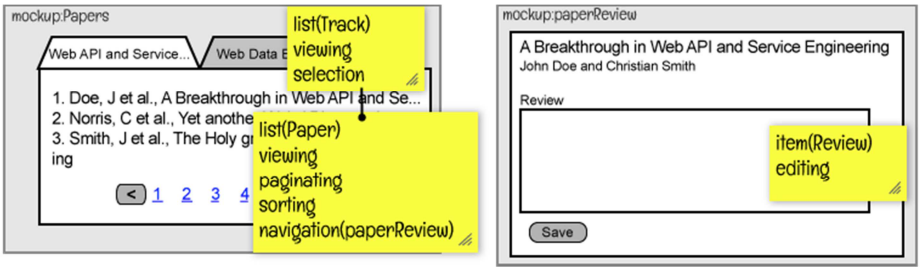

Fig. 2. Sample annotated mockups for a conference management system

Although there are other combinations of CRUD actions, in order to keep our approach simple, the two combinations described by our resource access patterns cover most actions used in web applications. If other particular combinations are required, a user story is added and the respective API has to be manually configured. Used with `Item`, `viewing` implies the content cannot be changed, while `editing` allows creating new instances, updating their content and deleting them. Used with `List`, `editing` additionally allows removing/reordering elements.

Associations. Since the structure of mockups can be arbitrarily complex, several content annotations can be present in a single mockup. Thus, MockAPI allows defining and relating different *Item* or *List* annotations. For this purpose, we introduce the concept of *Associations*. Each Association represents a directed relationship between two content annotations in the mockup and is graphically expressed by an arrow connecting them.

Sorting, Ordering, Filtering, Selection and Pagination. These 5 tags can only be applied to `List` to indicate it supports element sorting (e.g. by price), ordering (e.g. list prioritization using drag & drop), filtering elements (e.g. filtering by name), selecting elements (e.g. to apply some operation like deleting them) or pagination.

3.3.2 Dealing with Navigation

Navigation is another important aspect to define in web applications. It defines how interaction and data from the UI is fractioned and simplified in presentation units like pages, windows or menus, which can have an indirect impact in the API. For instance, a complex UI that displays a lot of data will be presented faster to the end-user if the API supports to get all the required information in a single request instead of many. This kind of relationship may be directly specified from one annotation to the other within the same mockup, as illustrated in Fig. 3a, where selecting a specific conference produces the tracks list to update. To relate data across two different mockups instead, an indirect navigation relationship can be defined between them, as shown in figure 3.b. To specify these navigations MockAPI includes the following annotation:

Navigation(DestinationMockupName). Indicates an element in the present mockup navigates to another mockup identified by *DestinationMockupName*, as seen in Fig.3b from *mockup1* to *mockup2*. Depending on the tooling used, the destination mockup can be identified by its name using different strategies like its filename.

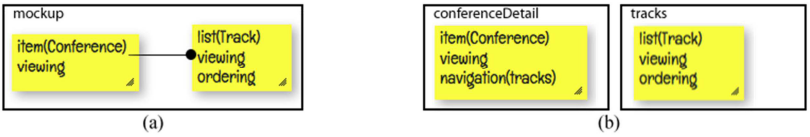

Fig. 3. Expressing relationships in annotations (directly or through a navigation)

3.3.3 Dealing with Custom Behavior

Features beyond manipulation of data objects and navigation are also considered in the approach. The underlying functionality cannot be generated automatically, but they can be modeled and added to the mockup to be implemented as separated user stories to be coded later, without breaking the annotation abstraction and requiring to make extensive language and code generation improvements. This kind of features can be introduced with the `SpecialFeature()` tag:

SpecialFeature(Description). Represents a complex feature that must be implemented in the API through direct coding, described in plain text (Description).

3.3.4 The MockAPI Metamodel

In order to abstract the structure of MockAPI annotations from their representations, we defined a detailed metamodel which structure can be observed in Fig. 4.

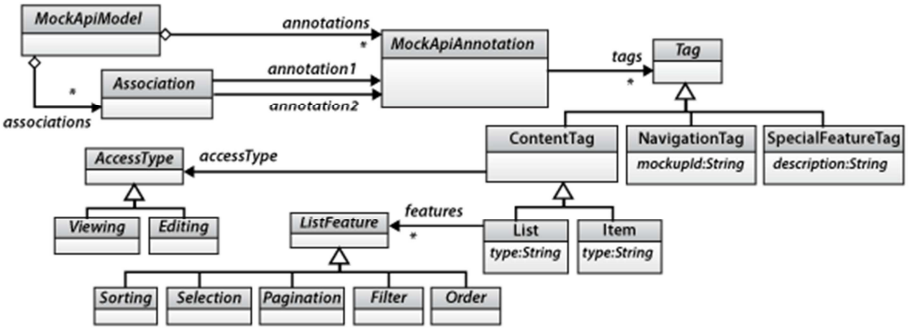

Fig. 4. Structure of the MockAPI metamodel

A MockAPI model (`MockAPIModel`) is composed by a list of annotations (`MockAPIAnnotations`) and associations (`Association`). An annotation is composed by a list of tags (`Tag`), which can be of type content (`ContentTag`), navigation (`NavigationTag`) or a special feature (`SpecialFeatureTag`) according to the types previously introduced. A `ContentTag` can be a `List` or `Item` and can have a specific `AccessType` (`Viewing` or `Writing`). In addition, a list can feature sorting (`Sorting`), selection (`Selection`), etc. A `NavigationTag` stores the id of the destination mockup and `SpecialFeatureTag` includes the description of the special behaviour to be implemented. Though not directly expressed in the metamodel, `MockAPIAnnotation` can only contain one instance of each `Tag` type but `SpecialFeatureTag`.

In 3.4 we describe how to generate API prototypes for set-based resources by analyzing content annotations, i.e. instances of the metamodel. The tags detailed in this section can be combined to form annotations placed over mockups. Fig. 5. shows a sample mockup of a conference manager with editable data of a conference, its editable and selectable tracks and read-only papers per tracks. Papers can be sorted and paginated. Clicking a paper navigates to another mockup called `trackDetails`.

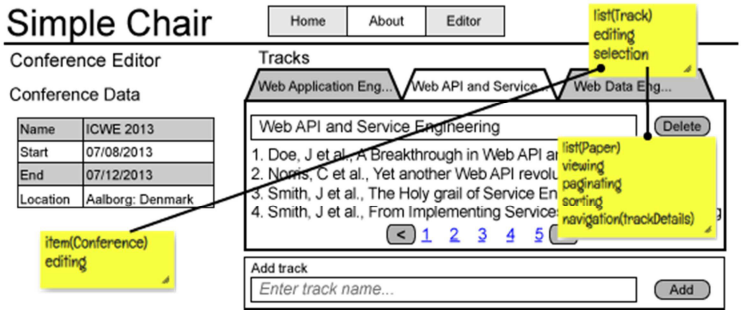

Fig. 5. Annotated conference manager mockup

3.4 Generating APIs from MockAPI

The MockAPI approach focuses on CRUD features of applications based on RESTful Web services; therefore, it constrains the supported design space. Providing guidance for these basic aspects supports agile developers in a frequent and time-consuming yet important part of work. Martin Fowler argues that "[d]isappointing as it is, many of the use cases in an enterprise application are fairly boring 'CRUD' (create, read, update, delete) use cases on domain objects" [17]. Any functionality beyond CRUD access to API resources, e.g., calculations, complex queries and statistical report generation, is handled in the usual agile way by creating a corresponding story. MockAPI simply sets the stage for developers to start implementing the missing functionality.

From an instance of our metamodel, the basic outline of the RESTful API can be inferred. Best practices for RESTful Web services [15] and the set-based navigation pattern [18] are applied to the modeling. The two central tags regarding content are `List` and `Item`. `List` tags are used to identify API resources and corresponding URIs. In the example shown in Fig. 5 `List(track)` implies the existence of:

```
/tracks
```

following the "Plural nouns and concrete names" principle described in [15]. Furthermore, tags defining user interaction aspects such as `Selection` and `Ordering` also influence the API. For instance adding a `Selection` tag in addition to the previous `List(track)` tag defines the items of the list, i.e. single `tracks`, to be individually selectable elements. Inferring resource URIs would additionally yield:

```
/track/<id>
```

This allows for access to the entire list as well as to a single item of the list identified by its id [15]. Although the same API can be achieved with `List(track)` and `Item(track)` – because to display a single list item it has to be identifiable in the API – the `Selection` tag additionally documents the user interaction requirement of selecting items from the list. The same applies to `Ordering`, which, only considering the API, is implied by `List(conference)` with access pattern `editing` as allowing update of a list implicitly enables reordering of its items. However, `Ordering` also specifies implementing list ordering at the application frontend e.g. by drag & drop.

Associations between content annotations are used to identify resource relationships explicitly visible in the UI mockups. For instance in Fig. 5, from `Item(conference)` and `List(track)` along with the association, i.e. the arrow from `Item(conference)` to `List(track)`, the following resource URIs can be inferred:

```
/tracks
/tracks/<id>
/conferences/<id>/track
```

It is important to note that MockAPI assumes a one-to-many relationship by default when `Item` and `List` are related. However, if an inverse one-to-many relationship is found in another mockup, the entire relationship is interpreted as many-to-many. Relationships between `Lists` are always assumed as many-to-many.

Further associations can be inferred even between annotations in separate mockups, using the `Navigation` tags. For instance, if an `Item(conference)` defines a navigation to a `List(track)` in a different mockup, a relationship between conferences and tracks will be inferred. In general, when annotations specify navigation to other mockups, the root content annotation is identified and an association is created between both content elements. The root content annotation of a mockup is an annotation with no incoming associations. If only one root annotation is present, the association is inferred automatically; otherwise it has to be refined manually.

4 Implementation

To assist the process, we devised tools that help through the main steps, as depicted in Fig. 6, starting from bare mockups to the generated API prototype.

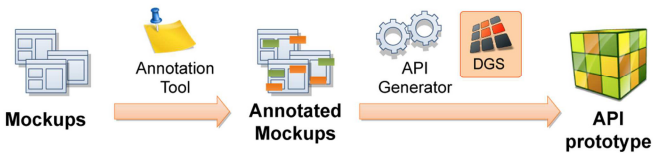

Fig. 6. MockAPI process with tooling support

In the following we explain process and tools for annotation automatic generation.

4.1 The Interactive Annotation Tool

While the structured annotations previously introduced can be applied manually over physical mockups to add semantics to the plain UI structure that they represent, semi-automatic API generation is not possible directly from them. To assist the annotation process and also to have a digital representation of the proposed annotations that can be used to generate the API, we developed a web annotation tool[8]. This tool allows importing any mockup image – e.g. hand-drawn or from image export capabilities present in mockup tools like Balsamiq[9] – and allows adding annotations over it. Fig. 7 shows a screenshot of the tool. During annotation, the tool parses the annotations to validate their structure and generates the underlying MockAPI model concepts.

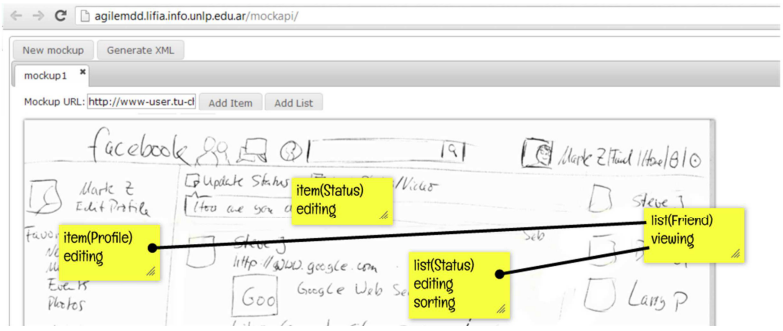

Fig. 7. Annotating a hand-drawn mockup with the MockAPI annotating tool

Once mockups have been correctly annotated, the tool provides a way of exporting an XML representation of the MockAPI model represented by the annotations. This model is used to further derive and configure the API automatically. Thus, the annotation tool works as the initial stage in the semi-automatic annotation-to-API process.

4.2 Generating APIs from MockAPI

In the following section we describe the implementation of a supporting tool that automatically generates a running API prototype from a set of annotated mockups by processing the XML representation of a MockAPI model. This tool applies the rules described in 3.4 to infer involved resources, access patterns and relationships.

4.2.1 WebComposition/DataGridService

In order to transform annotated mockups into a running API prototype, we employ the WebComposition/DataGridService (DGS) [19], which allows defining, creating and configuring resources at runtime and access via a RESTful interface. Our API Generator sets up the API prototype by configuring DGS XML resources.

---

[8] Available at: `http://agilemdd.lifia.info.unlp.edu.ar/mockapi/`

[9] Balsamiq Mockups - `http://www.balsamiq.com/`, last accessed 23-Feb-2013.

HTTP methods GET, POST, PUT and DELETE are supported on both resource and item level. Child elements of the XML root of the resource are treated as items of this resource, facilitating full read/write access to each of them separately. Additionally, DGS provides service and resource metadata maintained as RDF[10]. Configuration of the DGS and its resources is available through adding RDF statements to the metadata of the service or resource. Configuration on resource level includes the possibility to blacklist HTTP methods defining resource access policy. XML schema can be declared per resource to provide validation when HTTP-Requests attempt to modify the resource. On service level, relationships between resources can be declared consisting of source and target resource, a predicate, optionally an inverse predicate, source and target alias. Predicate is the RDF predicate to represent the relationship between items of source and target resource. Using inverse predicates, we leverage the benefits of RDF allowing DGS to automatically infer inverse relationships between items of resources related via (forward) predicates. To query items of target resource related to an item of source resource, target alias is appended to the source item path. Source alias works in the same way for inverse relationships.

Using the above set of DGS features we create a running API prototype at runtime.

4.2.2 API Generation

As shown in Fig. 8, API Generation consists of two phases: resource identification (1-5) and resource configuration (7-14). All types along with their access patterns are collected from items and lists defined in the MockAPI model (2-3). Relationships are identified processing associations and cardinality is determined as described in 3.4 (5).

Processing the derived set of types with access pattern information, the corresponding resources are created in the DGS, one per type (08). We pursue a set-based approach declaring the resources assuming containers of elements of the identified type. The container resource name follows the scheme `<TypeName>s`. While any occurrence of access pattern `editing` causes a type to be defined editable, only those types with all occurrences of `viewing` across all mockups are considered read-only. For each type identified read-only we configure DGS to restrict access to the corresponding resource accordingly denying HTTP methods POST, PUT and DELETE (10).

A default XML Schema is created per list (11) defining the root element matching the above name scheme and its content as sequence of elements named after the type, zero to unbounded occurrences. Currently, the content of the list elements is specified as *xs:any*, zero to unbounded, in order to allow for arbitrary data structures. However, the XML Schema can be easily adapted to incorporate specification of concrete data structures in future. For instance, a semi-natural language approach with statements like “A conference consists of name, location, startDate and endDate” is desirable.

Following the rules described in 3.4 relationships between resources are configured (14). Predicate names are created from a combination of resource names, e.g. `mkapi:ConferenceHasTracks` or `mkapi:TrackBelongsToConference`.

[10] RDF Primer - `http://www.w3.org/TR/rdf-primer/`, last accessed 29-Apr-2013.

```
01 foreach type Type with access Access in mockups
02   Types.Add Type
03   Accesses.Add (Type, Access)
04 foreach association (Source, Target) in mockups
05   Relationships.AddOrUpdate (Source, Target)
06
07 foreach type Type in Types
08   resource = DGS.CreateResource Type
09   if not Accesses.Contains (Type, "Editing")
10     resource.Deny [POST,PUT,DELETE]
11   resource.SetSchema DefaultXMLSchema(Type)
12 foreach relationship (Source, Target, Card)
13 in Relationships
14   DGS.DefineRelationship (Source, Target, Card)
```

Fig. 8. API Generation

Source and target alias are set to the resource name of the forward/inverse related resources. For instance `/conferences/<cid>/tracks` yields all tracks related to the conference with id `<cid>` via the `mkapi:ConferenceHasTracks` predicate. For the inverse relationship using `mkapi:TrackBelongsToConference` the generated URI path is `/tracks/<tid>/conference`.

5 Validation

In order to evaluate our proposed approach and identify potential shortcomings we conducted a brief validation. We tested the applicability of MockAPI in state-of-the-art websites by creating mockups for the most relevant user interfaces of 10 of the most popular websites based on the Alexa ranking [20]. To demonstrate the versatility of MockAPI, we used pen and paper mockups as well as digital mockup tools. The resulting mockups have been annotated using our interactive annotation tool and API prototypes have been generated using the MockAPI DGS API Generator.

MockAPI does not claim to create complete and mature APIs ready for productive use. Instead, we aim at providing a starting point for agile development by creating functional API prototypes. Therefore, an indirect metric is employed to evaluate our approach. We call this metric *coverage metric* and define it as follows:

Let M be a mockup and $P(M) = P_S \cup P_D$ the set of panels of M which provide user interface functionality. $P(M)$ can be subdivided into P_S, the set of panels which are static, and P_D, the set of dynamic panels. For instance, P_S includes navigation menus and buttons triggering predefined actions and P_D includes panels that dynamically depend on content or calculations such as lists of breaking news or displays of current time. Let A be the set of annotations added to M. Then $C(M) = \frac{|A|}{|P_D|}$ is the coverage metric of M. In other words, the coverage metric C is the ratio of

coverage of dynamic panels with MockAPI annotations. The main motivation behind this metric is to validate how much of the dynamic content can be modeled and further API generated automatically using the MockAPI infrastructure. Static content (P_S) is excluded from the evaluation as it is rendered directly, without making use of any API. Since some sites adjust static content to user preferences, we checked that panels remain the same for at least 3 different users to consider them as truly static.

We calculated C for each mockup of the popular websites used for validation. For the top 10 sites according to Alexa, we created 38 mockups and identified 150 dynamic data panels[11]. 134 of these panels could be covered by our annotations, which results in an average coverage metric of 89%. This indicates that the majority of dynamic panels in the most popular websites can be described using MockAPI annotations. Among those that could not be cover we identified 4 recurring groups: (1) results of calculations such as counting views, converting units etc., (2) results of foreign Web Service invocations such as weather information etc., (3) trending entities that are results of activity monitoring and access statistics such as trending news, tweets, hashtags etc. and (4) related entities that are results of similarity heuristics such as related articles, searches, news etc.

The high coverage for the rest of the panels shows that most features in the evaluated web applications can be specified as API operations. We found generated APIs to be surprisingly simple in comparison to the API and infrastructure of real web sites. However, since MockAPI is meant to speed up the development process, we argue that the functionality automatically generated from mockups is enough for the development team to start creating the application's front-end without wasting time coding the operations that the API must implement.

6 Conclusions and Future Work

We presented MockAPI, an approach based on mockup annotations which combines the advantages of agile and Model-Driven Development and demonstrated how to derive running API prototypes as starting point for agile development using our annotation metamodel. The brief validation indicated that MockAPI can cover most of the functionality found in the user interfaces of popular web sites.

In future work, we will focus on improving the ease of use and expressivity of our annotations. For instance, while currently annotations are simple lists of keywords, the proposed approach is a first step towards documentation and agile development support for technically less experienced stakeholders. Therefore, we want to evolve the annotation syntax to facilitate a semi-natural language description of UI elements and content in general and the structure of data in particular.

Moreover, we plan to extend the approach to cover additional aspects such as navigation, security or user interaction and consolidate the idea of constraint-based development with recent advances in mashup research to provide an environment for rapid development of web applications based on re-usable components.

[11] Analyzed data is available at `http://agilemdd.lifia.info.unlp.edu.ar/mockapi/validation`

Acknowledgments. This project is partially supported by the DAAD – MINCYT project 54367460 / DA/11/11.

References

1. VersionOne Inc.: State of Agile Survey (2011)
2. Rodríguez, P., Yagüe, A.: Some findings concerning requirements in Agile methodologies. Product-Focused Software Process Improvement 32, 171–184 (2009)
3. Leymann, F., Fritsch, D.: Cloud computing: The next revolution in IT. In: Proceedings of the 52th Photogrammetric Week (2009)
4. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE Computer Society (2008)
5. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33, 137–157 (2000)
6. Rivero, J., Grigera, J., Rossi, G., Luna, E., Koch, N.: Improving agility in model-driven web engineering. In: CAiSE Forum (2011)
7. Constantine, L.L.: Canonical Abstract Prototypes for Abstract Visual and Interaction Design. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 1–15. Springer, Heidelberg (2003)
8. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer 40, 38–45 (2007)
9. Kruchten, P.: Software architecture and agile software development. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, pp. 497–498. ACM Press, New York (2010)
10. Abrahamsson, P., Babar, M.A., Kruchten, P.: Agility and Architecture: Can They Coexist? IEEE Software 27, 16–22 (2010)
11. Eloranta, V.-P., Koskimies, K.: Aligning architecture knowledge management with Scrum. In: Proceedings of the WICSA/ECSA 2012 Companion Volume on - WICSA/ECSA 2012, p. 112. ACM Press, New York (2012)
12. Millard, D.E., Davis, H.C., Howard, Y., Gilbert, L., Walters, R.J., Abbas, N., Wills, G.B.: The Service Responsibility and Interaction Design Method: Using an Agile Approach for Web Service Design. In: Fifth European Conference on Web Services (ECOWS 2007), pp. 235–244. IEEE, Halle (2007)
13. Mukasa, K.S., Kaindl, H.: An Integration of Requirements and User Interface Specifications. In: 6th IEEE International Requirements Engineering Conference, pp. 327–328. IEEE Computer Society, Barcelona (2008)
14. Medrano, R.: Welcome To The API Economy. Forbes Online: CIO Network (2012)
15. Mulloy, B.: Web API Design: Crafting Interfaces that Developers Love. Apigee (2012)
16. Schwaber, K.: Scrum development process. In: Proceedings of the Workshop on Business Object Design and Implementation at the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1995) (1995)
17. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2012)
18. Rossi, G., Schwabe, D., Lyardet, F.: Improving Web information systems with navigational patterns. Computer Networks 31, 1667–1678 (1999)
19. Chudnovskyy, O., Gaedke, M.: Development of Web 2.0 Applications using WebComposition/Data Grid Service. In: The Second International Conferences on Advanced Service Computing (Service Computation 2010), pp. 55–61. Xpert Publishing Services (2010)
20. Alexa: Alexa Top Sites, `http://www.alexa.com/topsites`