juice

Daniel Cowgill and Graham Lowe

February 3, 2009

1 User Interface Framework

1.1 Background

In 2004, Community Connect Inc. started a project to rewrite their entire code-base. We finished this initiative in late 2007. The end-result was a software architecture that facilitated concurrent development of the database and domain logic layers of their application. Despite productivity gains in the *backend* layers of the system, *frontend* (i.e., user interface) development was still arduous.

1.2 Problem

To help you understand the deficiencies that existed with user interface development, we will describe the typical workflow for developing a new interface. But first we will introduce the roles involved in the workflow[1]:

- The *product designer* specifies initial requirements for a product.
- The *interaction designer* translates initial requirements to user interface requirements, often expressing them as wireframes (i.e., a low-fidelity paper-based graphical specification for the product).
- The *frontend developer* interprets user interface requirements from wireframes, creating a non-functional HTML user interface prototype.
- The *backend developer* interprets requirements, creating a database model and writing domain logic code to implement the product's functionality; she also integrates this *backend* with the HTML user interface prototype, resulting in a functional product.
- The *quality assurance specialist* tests the functional product, ensuring that it is consistent with the product specification.

Now that we are familiar with the roles, we'll describe a typical workflow for developing a user interface; this workflow is depicted in Figure 1.

The product designer creates rudimentary requirements for a product and hands them off to the interaction designer. He uses these requirements to create wireframes, which he then gives to a frontend developer who uses a proprietary user interface prototyping tool to generate an HTML-based prototype. The frontend developer disseminates this generated HTML to the backend developer who is responsible for integrating this HTML with PHP[1] code to create a functional product. Once the product is functional, it is ready for inspection by the quality assurance specialist. If the QA specialist finds user interface defects, the issue will be sent to the frontend developer who then revises the HTML, which must then be reintegrated by the backend developer. This cycle repeats until the product is deemed perfect.

There are several issues with the workflow we've described. The frontend developer wastes time creating an HTML prototype of the user interface. Once created, the prototype serves as a specification of the product's HTML, so subsequent modifications to the HTML must first be made by the frontend developer

[1]we've omitted roles (e.g., project manager) that are non-essential to understanding the workflow problem.

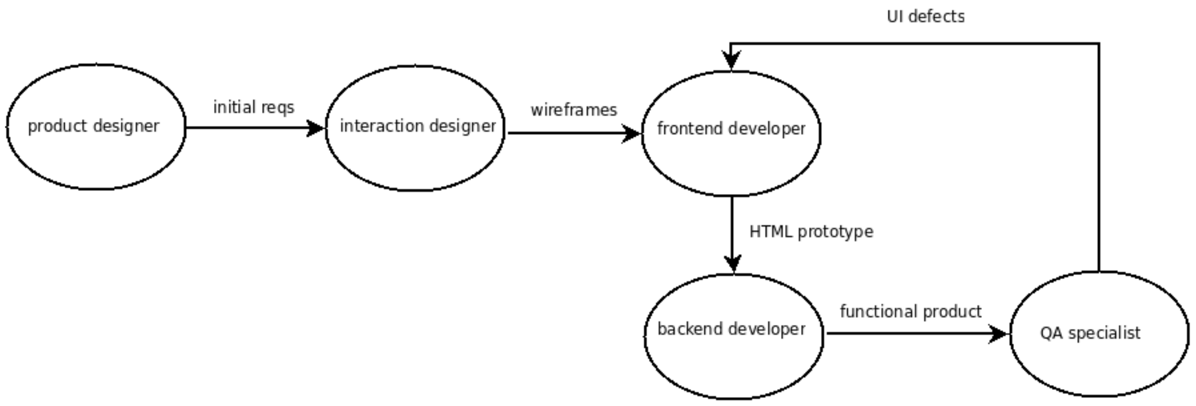

Figure 1: Inefficient UI development workflow

to the prototype and then translated to the functional product by the backend developer. It would be better if there was a single version of the HTML, preferably in the end product. Another deficiency is that the QA specialist must wait for the backend developer to finish his work before she can identify problems that are only apparent when working with a functioning product. Furthermore, correcting these defects often result in changes to the data model and domain logic code, so that in addition to any HTML reintegration, the backend developer will have to make changes to the backend layer as well. If the QA specialist could test the functional product earlier in the process, then less rework would be required. Another issue with the process is that frontend developers use a high-level language to describe the user interface, treating HTML as a low-level target language. Backend developers must deal with this generated HTML—this HTML is not designed for human comprehension—making integration with PHP code more difficult.

It also worth mentioning that because of the sequential nature of this workflow, it is futile to add frontend development resources without adding backend development resources and vice-versa. An improved workflow would allow concurrent frontend and backend development.

1.3 Towards a Solution

Before designing the user interface framework, we first analyzed the issues with workflow and determined that many of the problems could be alleviated by enforcing a clear separation between the *frontend* and *backend* development tasks. Given this prerequisite, we set about defining other essential requirements for the framework. To save the wasted time of creating prototypes and to allow the product designer and QA specialist to provide feedback earlier in the workflow process, we determined that the framework should be able to produce working user interfaces without a functional backend. We also liked the idea of using a high-level user interface description language. Also, with the popularity of smart interfaces like Gmail [2], we added the requirement that the framework should have good support for AJAX.

With these initial set of requirements, we started thinking about software engineering issues. It was important that the framework allow the creation of arbitrarily complex user interfaces while ensuring that the code to do so wasn't overly complex; to achieve this goal, we decided that the framework should allow interfaces to be built by combining several simple components into a single component.

1.4 Prototyping

Armed with these requirements, we decided that we needed to prototype some solutions to gain familiarity with the technical challenges.

We wrote our first framework prototype in PHP. This version of the framework required the client programmer to specify the user interface as a series of PHP objects which map to generated HTML and JavaScript. Some of the benefits of using PHP were:

- Backend programmers were already familiar with language.
- PHP, because it was initially designed as a templating language, allows HTML to be easily generated.

However, we also found some drawbacks:

- The programming model was very complex because PHP code is executed in the server and JavaScript code is executed in the browser.
- Because the backend code was also written in PHP, using PHP as a frontend language invites bleed through of non-user interface code to user interface code and vice-versa.

We soon realized that PHP wasn't a good fit for the framework, so we decided to write our second prototype in pure JavaScript. Some of the upfront benefits were:

- Frontend programmers are familiar with language.
- JavaScript is arguably a more powerful language than PHP.
- AJAX support is built into the language.
- Enforces a clear separation of concerns because the frontend is written in a different language than the backend.

Here were some of the drawbacks:

1. JavaScript, or rather browser implementations of JavaScript, are notorious for being quirky.
2. Although some of the programmers were familiar with JavaScript, the sophistication of the framework would require deeper knowledge.

After prototyping, we found ourselves enamored with JavaScript. Although it is much maligned, at its core it's a great language. We found its combination of first class function, closures, and uniform syntax to be very powerful. Furthermore, we found that there are several open source libraries that hide the need to write special workaround code for browser quirks.

1.5 juice

Once we had finished prototyping, we set out to work on our framework in earnest. In this section, we describe the result of our efforts—a framework we called *juice*, which stands for **J**avaScript **U**ser **I**nterface **C**ompiled **E**nvironment.

Some qualitative goals we tried to achieve when creating juice were: making it as declarative as possible, keeping the core concise, minimizing complexity, and ensuring its components were orthogonal.

While juice is a framework, it can also be thought of as a language for describing a user interface, so we'll explain it as such—by enumerating its primitives and means of composition. Figure 2 illustrates juice's architecture.

1.5.1 Primitives

- A *RPC* represents a remote service that is implemented by the backend. RPC definitions are written by a backend programmer and specify the expected input and output values for an RPC function call. RPC definitions are fully declarative. Here's an example RPC definition:

  ```
  juice.rpc.define(
    {name: 'add_bulletin',
     service_name: 'add_bulletin',
     req_spec: {subject: 'string',
                body: 'string'},
     rsp_spec: null});
  ```

  RPCs are invoked by widgets. Because of their asynchronous nature, the caller must supply a

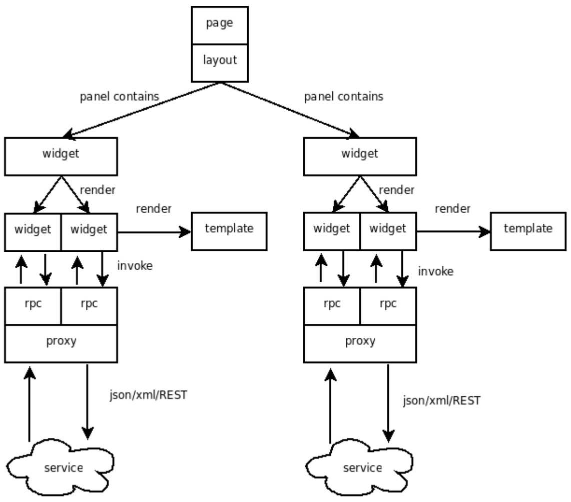

Figure 2: juice

callback function which is passed the response from the backend service. Here's an example of how to call an RPC:

```
proj.bulletins.add_bulletin(
  {subject: "Lunch",
   body: "Free lunch for everybody!"},
  function(rsp) {
     // Do something with response
     ...
  });
```

Each RPC invocation is handled by a proxy. This layer of indirection can provide several benefits. For example, a proxy can package several RPCs into a single network request, reducing the number of network round-trips. Or a mocking proxy may be used to fake a backend service, allowing the user interface to be almost completely functional before the backend service is even implemented. In either case, the proxy behavior is completely transparent to the caller of the RPC.

- *Widgets* describe user interface behavior and presentation. Widgets can call RPCs to retrieve or manipulate data. A widget's presentation is specified via its template.

  Here's an example widget definition:

```
juice.widget.define(
  'username_and_photo',
  function(that, my, spec) {
    var info = {
      username: spec.username,
      username_url:
        proj.pages.user.url(spec),
      image_url: spec.image_url
    };

    my.render =
      templates.user_and_photo(info);
    });
```

- A *template* defines a widget's HTML presentation through a custom templating language that is compiled down to a JavaScript function that is optimized for rendering speed.

  Here's an example template:

```
<form disabled="disabled">
  {% if label %}
  <legend>{{label}}</legend>
  {% endif %}
  {% for k, input in inputs %}
  {{input}}
  {% endfor %}
  {% for k, button in buttons %}
  {{button}}
  {% endfor %}
</form>
```

1.5.2 Means of Composition

- *Widgets* can be composed from other widgets. This feature allows client programmer to create widgets of arbitrary complexity through composition while keeping the code as simple as possible.

  Here's an example of a composite widget (composed from 5 smaller widgets):

```
juice.widget.define(
  'my_form',
  function(that, my) {
    var form =
      proj.widgets.form({submit_label: ’search’})
       .add(’gender’,
            proj.widgets.gender())
       .add(’orientation’,
            proj.widgets.orientation())
       .add(’age_range’,
            proj.widgets.age_range_input())
       .add(’extra’, w.extra());

    my.subscribe(
      form, ’submit’,
      function(data) {
        juice.log(juice.dump(data));
      });

    my.render = form.render;

    form.init();
  });
```

- A *page* definition specifies how a set of widgets are to be placed on a web-page using a layout definition.

  A layout specifies the panels that appear on page. Here’s an example of a layout definition that defines a top panel, two middle panels, and a bottom panel:

```
juice.layout.define(
  ’two_column’,
  {top: null,
   middle_: {middle_left: null,
             container_:
               {middle_right: null}},
   bottom: null});
```

  And here’s an example page definition:

```
juice.page.define(
  {name: ’bulletins’,
   title: ’Bulletins’,
   path: ’/bulletins/’,
   layout: proj.layouts.two_column,
   widget_packages:
     [’bulletins’, ’message_center’],
   init_widgets: function(args) {
     return {
       top: [
         w.core.login_assertion(),
         w.core.global_nav()],
       middle_left:
         [w.message_center.nav()],
       middle_right:
         [w.bulletins.uber_inbox()]};
   }
  });
```

1.5.3 Packaging and Compilation

To support large-scale software development, juice requires organizing widgets and RPCs into packages. Packages allow the client programmer to define widgets and utility functions that are hidden. Juice also requires a compilation step to create a functional user interface. During this build step, the following tasks are performed:

- Source code is checked for syntax errors and software engineering errors (e.g., cyclical dependencies).

- Juice programming constructs are compiled into appropriate JavaScript functions and HTML. For example, templates are compiled into JavaScript functions that are optimized for rendering performance and page definitions are compiled into a corresponding HTML representation.

- RPC and widget packages are compiled into source files that are scoped such that local function and widget definitions are hidden. Also, dependency information is extracted so that the resulting HTML pages include a minimal amount of external JavaScript files.

- JavaScript is optimized into its most compact representation to reduce the amount of bandwidth required to deliver a file.

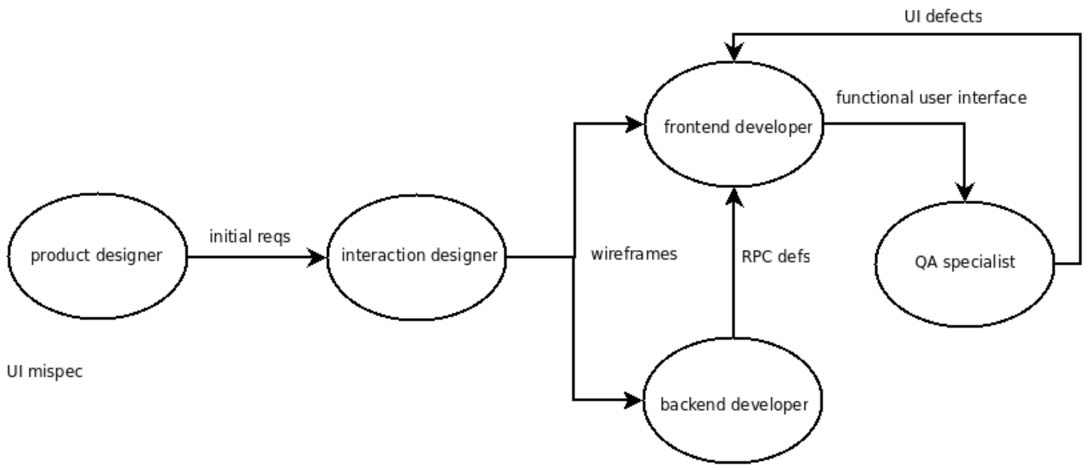

Figure 3: Improved UI development workflow

1.6 Improved Workflow

Juice provides a clear separation between frontend and backend development tasks and the ability to *mock* backend services to create a standalone functional user interface. This allows us to define an improved workflow which is illustrated in Figure 3.

In this workflow model, changes to the user interface don't require a integration work by the backend developer. Not only does this remove a serial step in the original workflow, but it allows backend and frontend resources to be added to a project independently. Furthermore, because a functional user interface can be produced for a product without a functional backend, user interfaces can be tested before the backend is completed. Another benefit of this approach is that it allows multiple experimental user interfaces to use the same backend services.

1.7 Status

While the juice framework is complete[2], it has yet to be used in Community Connect Inc's production user interface code. This daunting task has been planned for my Spring semester internship.

The core of the juice framework is approximately 4,200 lines of JavaScript (including whitespace and comments), 300 lines of Python, and 50 lines of the custom template language. We also produced a juice demo application, a product similar to Google's Gmail. This application consists of approximately 3,250 lines of JavaScript and 1,050 lines of the custom template language.

[2]We still have to write end-user documentation.

References

[1] http://www.php.net/

[2] http://www.gmail.com/