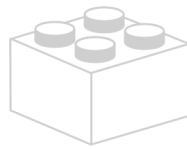


BRICKS



Prioritizing innovation and agility for
modern, native, server-driven mobile apps.

BRICKS is a new paradigm in mobile app development, which will cut our time to market for new apps from months to weeks, and cut time for feature updates to existing apps from weeks to days, setting a new standard in speed, quality, and innovation for fully native, mobile apps.

To execute our shift in focus to mobile, we need to do more—faster, better, and cheaper. And yet, for any amount of money, we are unable to keep pace with our ambitions, to maintain and release new features in our core apps quickly enough, even to find enough qualified personnel to staff those core and new products.

BRICKS will enable us to move from our current state of spending the vast majority of our time and resources on maintenance and technical development, to a future where the bulk of our time is spent on product design and editorial development. In other words, BRICKS will allow us to focus on what we're building and why, not on how we're building it.

We'll do this by creating a reusable set of components, or “BRICKS”, which can be recombined in any number of ways to create any number of apps, on any number of platforms.

Thinking about our flagship newsreader: at its core, it is an application to present different kinds of views to the user, mainly text and multimedia, and to allow that user, by tapping on different pieces, to navigate to screens filled with more of these views.

BRICKS is predicated on identifying the atomic elements of our apps, perfecting those, and moving control of recombining them from our apps, which are fragile, rigid, and unforgiving, to our servers, which are cheap, plastic, and completely within our control.

Because every app is nothing more than a combination of bricks, and because the composition of those bricks is controlled entirely from our servers, we can fundamentally innovate and change our existing apps easily, for all users, immediately. And, better, each of those changes

can be uncoupled from others in flight, avoiding the domino effect of a bug in one shipping feature holding countless other features hostage.

Creating new features and apps that are different combinations of text, images, and multimedia then becomes a simple design exercise, not a technical one, since every app, using BRICKS, knows how to natively display any combination of any bricks it understands.

And, because BRICKS will be built as an open standard, anyone can create new bricks.

Expanding that vocabulary to experiences we haven't yet imagined becomes an investment in all our digital futures: every investment we continue to make in our mobile apps adds even more flexibility, broadening our palette, as we continue to build a repertoire of reusable lego pieces, available to any of our current and future applications.

As part of our work on the mobile imperative, we've begun this work, and built a working prototype that consumes NYT's daily feeds, and presents a completely personalized newsreader. This is not a fantasy ideal too good to be true, but rather a very achievable future that could yield better products, lower costs, and true leadership in mobile.

Background: Mobile development today

Despite our best efforts to innovate our existing products and create new ones, our biggest obstacles have been technical ones. The costs, both in personnel and in time, to create and modify experiences are too great.

In our core newsreader app, for example, a change as simple as modifying a font, or changing the order of stories can take months, even in the best of scenarios.

Let's assume for this example that we want to change all Opinion headlines from the regular Cheltenham typeface to a Cheltenham Wide Italic, something that should take a native app developer, let's say, between a few minutes and a few hours. An afternoon's work, at best.

1. Once our font change makes it out of the backlog and onto a developer's todo list, the native app developer implements the change.
2. The code is then peer-reviewed.
3. The changes are merged into a development alpha release, and reviewed by the designer, who may or may not kick the ticket back to the developer to make further changes.

4. Once both are satisfied, those changes must be folded into a native app release, which happens, at best, every month.
5. The inclusion in the very next release of a completed feature, no matter how trivial, is far from assured: because there is such high risk of instability, features are often excluded to limit the amount of change going out at any one time, or releases cancelled or postponed due to high-priority production issues that must be "patched", diverting developer, QA, and product attention.
6. In practice, patch releases can push our simple font change another month or more away from user's hands, as it waits for the next batch of features to be compiled into a new release.
7. Once our feature does make the cut to be included, our QA team spends between several days and a week testing all features of a release, most of which will be unrelated to our change. Any bugs are triaged, and either deprioritized or fixed, which can push the release out still further. If, at this point, problems with our feature are found, it is extremely likely for that feature to be scrapped, or held for yet another release, yet another month away, at best.
8. Assuming, however, that we do make it through QA, the release then goes, in the case of iOS, to Apple for review. This process takes between a handful of days, or, more likely, one to two more weeks, until the app is ready to be released. Again, at this stage, Apple can, and does, reject the version of the app for any reason, likely reasons unrelated to our font change and, sometimes, unrelated to changes made in this release at all. At this point, we go back to the "patch" phase (step 5).
9. Assuming Apple approves, the app is released to the App Store, and it takes several days for the bulk of our users to upgrade their software, one download at a time.
10. If all goes well, our font change was successful. However, if there are any issues with our changes at this point, they are stuck in the compiled bedrock of our apps, stuck on millions of smartphones all over the world, unchangeable. Suppose the developer accidentally forgot to make the font change on Krugman's blog posts, as they technically stem from a different parent section: fixing that error requires going all the way back to step one, and could take several more weeks (at best) or (more likely) months.

The above scenario is far from uncommon: in fact, it would be considered uneventful. This is how, for a feature as trivial as changing a font, it takes one to six months to go from a product priority to a product reality.

This is why, when we talk about events like elections, the World Cup or the Olympics, we must begin planning many, many months ahead of time, and, ultimately, have favored HTML,

web-based solutions: native development has been, at least until now, too inflexible, too slow, too expensive, and, with the difficulty of attracting and managing top native talent, too difficult to truly innovate with or react quickly to breaking news.

Our experiments with HTML5-based development still show that experience to be sub-par. Ask any experienced Javascript engineer, and they will tell you that, while it is possible to get the web to do interesting things, and to feel something like a native app, turning that last corner to a first-class user experience is still, even today in 2015, extremely difficult and elusive.

As our own Alastair Coote, formerly of NYT's Mobile Web team, and now writing best-in-class interactive news experiences downstairs in the newsroom, recently wrote on his blog on the launch of new web technologies in iOS 8:

I've been playing around with embedding WebViews in native apps for some time now, but I've never managed to get to them to a point where I'd be happy to use them in an app people would actually use. The bad reputation is justified – aside from anything else, they're just slow . . . I hope that in time WKWebView will be the saviour we've all been waiting for. But it isn't yet.

<http://blogging.alastair.is/the-disappointing-reality-of-wkwebview/>

A new way

BRICKS is a new way of thinking about native mobile development that prioritizes quality, speed, and agility.

By rethinking the way we write mobile software, we'll be able to deliver fully-native apps that are flexible enough to launch brand new features to users in days, not weeks or months, freeing up our multi-million-dollar investment in mobile infrastructure to innovate, not simply maintain.



We don't write a web browser when we deliver our news to our website. We instead format our content in an agreed upon way, so that web browsers can present our articles with the reusable pieces that they have committed to support

In the same way, BRICKS is a cross-platform standard to format our content with reusable pieces, so we can focus less on reinventing the wheel, and more on refining the products and journalism that define this company in the eyes of the world.

Proof of Concept

Our early proof of concept created native section front and article pages from just two components: an image component, which could take any shape or size, and a text component, which could be flexibly styled to create headlines, bylines, and article paragraphs.

Both components could be stacked, reused, and recombined to create the sections and screens that make up an app. All of this is dictated by a single feed. In essence: the app is the feed.



Building off of that early success, we've developed a library of components that drive a fast and flexible prototype that has been delivering the news to roughly 60 beta testers worldwide since late March 2015.

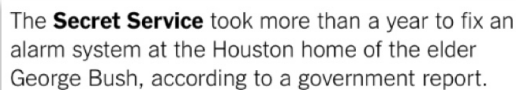
The following section describes the basics of some of those components, as well as how we've combined them to realize our product and design ambitions faster than we ever have before.

BUILDING BLOCKS

Text

Text is at the heart of everything we do, and BRICKS has first-class support to place and style text with granularity and ease.

Take the example of displaying this watching post's card, with the visual results on iOS displayed on the left, and the JSON object that produced it on the right:



The **Secret Service** took more than a year to fix an alarm system at the Houston home of the elder George Bush, according to a government report.

```
{
  "treatment": "text",
  "textItems": [
    {
      "text": "The "
    },
    {
      "text": "Secret Service",
      "style": "watching_bold"
    },
    {
      "text": " took more than a year to fix"
    }
  ],
  "style": "watching_post_body"
}
```

A few things to notice:

- the JSON specifies that this item will represent text via its `treatment` field
- the item specifies an *array* of `textItems`, with one more objects inside. This represents the actual characters that will be rendered to the screen.
 - Each entry in the array can specify a different style, which allows us to mix styles and provide rich-text support, as displayed above.
 - If no style is specified, the default style of the text object is used.
- every text object specifies a `style` field (in fact, as we'll learn, most objects specify a style field, even if they contain no text). This controls, as you'll read in the following page, everything from the font that should be used, to line height, to insets and background color.

BUILDING BLOCKS

Styles

BRICKS uses styles, similar to CSS, to configure and place text and other elements. Because this formatting is part of the feed itself, it's possible to completely change the look and feel of every element, at any time, without any other code changes.

Let's examine the style entries for that watching post:

The **Secret Service** took more than a year to fix an alarm system at the Houston home of the elder George Bush, according to a government report.

```
{
  "watching_post_body": {
    "font-name": "NYTFranklin-Light",
    "font-size": 15,
    "alignment": "left",
    "line-height": 1.2,
    "top": 10,
    "left": 10,
    "right": 10,
    "bottom": 10
  }
  "watching_bold": {
    "parent": "watching_post_body",
    "font-name": "NYTFranklin-Bold"
  }
}
```

These should look familiar: they specify the font we're using, the size, alignment, and line height. Note that styles can inherit from one another: `watching_bold` has all the same formatting as `watching_post_body`, but it changes the `font-name` field to `NYTFranklin-Bold`.

Typographic style options

You'll find these options particularly useful when styling text:

font-name	The font that should be used
font-size	The size for that font
line-height	The line height, in ems. e.g. 1.2
alignment	left, right, or center
paragraph-spacing	The space after each paragraph.
paragraph-spacing-before	The space before each paragraph.

BUILDING BLOCKS

Styles (cont.)

General style options:

The following options apply to text elements, but are equally useful when placing and styling other kinds of bricks:

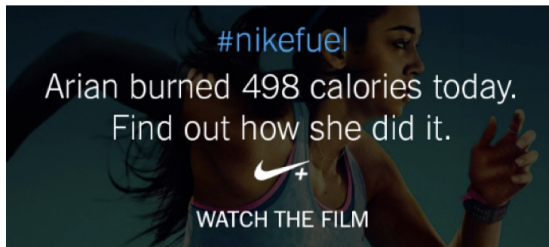
parent	The name of a parent style that this style should inherit from.
top	The inset of this element from its top edge. Like padding-top in css.
left	The inset of this element from its left edge.
bottom	The inset of this element from its bottom edge.
right	The inset of this element from its right edge.
color	Can be hex or rgba. #f3f3f3 / rgba(0,0,0,0.2)
bg	The background color for this element. Hex or rgba.
rules	<p>Rules that should be applied to this element. If defined, this entry should contain keyed objects for “top”, “bottom”, or both. For example:</p> <pre>"top_rule": { "rules": { "top": { "color": "#cccccc", "left": 0, "right": 0, "height": 0.5 } } }</pre>

As you'll see, styles are an integral part of every other element.

BUILDING BLOCKS

Images

Fast and performant images are essential, for everything from interface elements like logos and buttons, to news photographs, to, as in the case below, advertisements.



```
{
  "treatment": "image",
  "image": {
    "url": "https://s3.amazonaws.com/nytm2/
    "height": 320,
    "width": 720
  }
}
```

As with the text item example, this item declares itself an image via its `treatment` field. Beyond that, the only requirement is the `image` field, which itself contains the url where the image can be loaded from, a height, and a width. This sizing information is essential, because it lets us size the views where these images are going to go in advance of loading the actual image.

Although this image doesn't specify any, images can use the `style` field to inset their images, or set their background color (clear, by default) while the image loads.

BUILDING BLOCKS

Stacked Items

Many of your components will be combinations of existing ones. Stacked items are built up of other elements, vertically stacked on top of one another to build a cohesive whole.

For example, our article cards in the prototype are actually built from three separate elements: an image item and two text items.



```
{
  "treatment": "stacked",
  "style": "topRule",
  "items": [
    {
      "treatment": "image",
      "image": {
        "url": "http://static01.nyt.com/image",
        "height": 507,
        "width": 768
      }
    },
    {
      "treatment": "text",
      "style": "fullBleedHead",
      "textItems": [
        {
          "text": "James Spader Prepares for"
        }
      ]
    },
    {
      "treatment": "text",
      "style": "fullBleedSummary",
      "textItems": [
        {
          "text": "The actor, who is starring"
        }
      ]
    }
  ]
}
```

Note that, again, the item specifies what kind it is through its `treatment` field, specifies optional styling, and then includes an array of `items`, each of which is, itself, its own complete brick.

This will be a common pattern: by reusing some components to build others, we're able to move incredibly quickly and efficiently.

BUILDING BLOCKS

Screens

An app is made up many screens of content. Those screens, in turn, are built up of the items we've already learned about: images, text, and more.

Put another way: apps are built from screens, screens are built from items. And, as we'll soon learn, any item can perform any number of events, one of which is linking to any other screen. It is the links between all of these items that make up the complete app experience.

Let's take a look at an example screen:



```
"home" : {
  "treatment": "front",
  "style": "tan_sectionfront",
  "header_view": {
    "image": {
      "url": "https://s3.amazonaws.com/nytm2/",
      "width": 750,
      "height": 108
    },
    "treatment": "full_bleed_photo"
  },

  "refresh": {
    "url": "http://m2feed-dev.elasticbeanstalk.com/m2feed-dev",
    "method": "POST",
    "preferences": [
      "favorite",
      "queue",
      "onboarding"
    ]
  },
  "left_button_item": {
    "treatment": "image",
    "image": {
      "height": 18,
      "width": 15,
      "url": "https://s3.amazonaws.com/nytm2/"
    },
    "style": "barButton",
    "events": [
      {
        "id": "sectionList",
        "action": "present_screen",
        "event_type": "tap"
      }
    ]
  },
  "items": [
    . . .
  ]
}
```

BUILDING BLOCKS

Screens (cont.)

Let's take each element in turn:

- Note that every screen has a unique identifier, and is keyed by that identifier in a node of each app configuration called “screens”. In this case, this screen’s identifier is “home”. Other screens and items can use this identifier to link to, present, or embed content from this screen.
- The `treatment` element. Possible values include `front`, for screens representing section fronts / index pages, or `detail`, for screens representing articles.
- The `style` element, which applies styling information to your screen. The background color, for example.
- The `header_view` field allows you to place any item type in the navigation bar. Here, we’ve placed The Times’ nameplate there, taking advantage of the image brick.
- If your screen is refreshable, you may include instructions for how that happens within the `refresh` element. On iOS, for example, this would allow your user to pull to refresh the page and, upon success, would replace the contents of your screen with the home screen of the app configuration fetched from the server.
- You can assign any type of item to be the left or right buttons on the navigation bar, via the `left_button_item` and `right_button_item` fields. Here, we’ve chosen an image. Note that this example introduces a new field to that object: `events`. More to come on that in the next section.
- Lastly, every screen has a field called `items`, which is an array with zero or more items to display on that screen. These items are stacked on top of each other to make up the visible content displayed to the user. *Note that to save space, items are omitted from this diagram.*

```
"home" : {
  "treatment": "front",
  "style": "tan_sectionfront",
  "header_view": {
    "image": {
      "url": "https://s3.amazonaws.com/nytm2/
      "width": 750,
      "height": 108
    },
    "treatment": "full_bleed_photo"
  },
  "refresh": {
    "url": "http://m2feed-dev.elasticbeanstal
    "method": "POST",
    "preferences": [
      "favorite",
      "queue",
      "onboarding"
    ]
  },
  "left_button_item": {
    "treatment": "image",
    "image": {
      "height": 18,
      "width": 15,
      "url": "https://s3.amazonaws.com/nytm2/
    },
    "style": "barButton",
    "events": [
      {
        "id": "sectionList",
        "action": "present_screen",
        "event_type": "tap"
      }
    ]
  },
  "items": [
    . . .
  ]
}
```

BUILDING BLOCKS

Events

Events, usually triggered by some kind of user interaction with an item, have two key elements:

- an `event_type`, which defines the event's trigger (e.g. tap, long-press)
- an `action`, which describes what should happen when the event fires.

Returning to the stacked item example, here it is with the `events` node added:



```
{
  "treatment": "stacked",
  "style": "topRule",
  "items": [ ... ],
  "events": [
    {
      "event_type": "tap",
      "action": "push_screen",
      "id": "100000003641062"
    }
  ]
}
```

Here, we can see that, on tapping the view created for this item, a new screen should be pushed. Events also may contain arbitrary fields with specific information to help perform their action. In this case, the node includes an `id` field, which is the id of the screen* to push.

Multiple events can be added to any item, even events with the same `event_type`. Pushing a screen, for example, could be combined with toggling a setting, or presenting an alert.

** One ancillary benefit to this approach: while the same screen can be pushed by multiple events on multiple other items, only one copy of that screen actually exists. That is in contrast to our content-feed based approach, wherein when an article belongs to more than one section—Top Stories, and Most Emailed, and Opinion, for example—multiple copies of that article are downloaded with each update.*

BUILDING BLOCKS

Events (cont.)

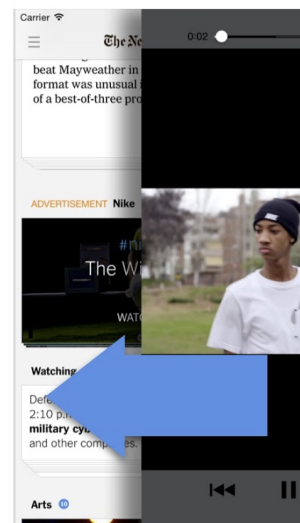
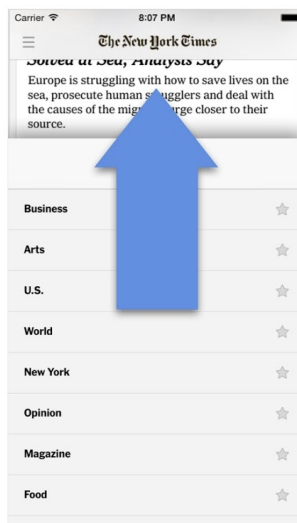
A few of the events that we have come to rely on:

- The `push_screen` event pushes a screen onto the existing navigation stack.
- The `present_screen` event presents a screen modally. Note that the screen that was presented is then free to push or present its own screens which, incidentally, is how we built our section navigation.
- The `append_value` and `remove_value` actions allow the app to store arbitrary information in arbitrary categories on demand. It's how we built the favoriting functionality in our prototype.
- The `open_url` action opens a URL*.
- The `alert` action presents an alert to the user.
- The `share` action shares a piece of text or URL via the app's native sharing capabilities, —email, SMS, Facebook or Twitter, for example.

```
{
  "event_type": "tap",
  "action": "present_screen",
  "id": "sectionList"
}
```

```
{
  "event_type": "tap",
  "action": "push_screen",
  "id": "10000003641062"
}
```

```
{
  "event_type": "tap",
  "action": "open_url",
  "url": "https://www.youtube.c"
}
```



** Interesting to note: when we were building the native ad stacks into the prototype, we wanted the YouTube videos to play immediately on open, so we built that capability into the `open_url` action. That capability now exists for any URL, including URLs for our own content. That is the power of BRICKS: everything we do makes it easier to do more things.*

Conclusion

If we're going to compete in mobile innovation, and if mobile is strategically important to The Times' future, we're going to need to be able to try more things, fail more, and learn more quickly from those failures. A server-driven UI like BRICKS is the only feasible way forward.

We are not in a position to experiment and take risks in the ways we need to to succeed in mobile today. While increasing staffing is an option if we can find the people, adding those people won't necessarily make us move more quickly, and it certainly won't be cheap. If we could conceive of ten sound new editorial products this year, we couldn't afford to launch them the way we have in the past. We'd have to place a bet on one or two. With BRICKS, we could place many, many more bets.

Developing BRICKS SDKs across the platforms that are important to us—iOS, Android, the web and mobile web—will not only allow us to move more quickly in our existing products. It will allow us to pursue products on those platforms that we might not otherwise have thought economically or logistically feasible. It will allow us to share the design DNA of our products across those platforms, as we cross pollinate capabilities between them.

Everything we do in technology—from carefully considered product design, to hand-coded custom interactives, to breaking news, to dramatic personalization and A/B testing, becomes easier by orders of magnitude when developing with BRICKS. Because the entire app experience is plastic and delivered server-side, different users can have vastly different apps. The same native codebase can drive many different products. With a single developer and a few weeks, it's possible to make highly detailed live-data prototypes with BRICKS, especially if we invest in complementary reusable server architecture to power these apps. Static, tappable, readable prototypes can be assembled, by hand, in a matter of hours.

In the near term, BRICKS can make our news presentation more flexible. In the long term, committing to a technology like BRICKS could be as important as the move from the hand-coded, HTML web pages of the 1990s, to CMS driven frameworks that have enabled so much of our success in the desktop world. BRICKS will do the same on mobile, not only as a means to flexibly publish our report, but as a means to flexibly publish our products themselves.