[DRAFT] Valkyrie: Event Listing Page Proposal

Author: Jesse Dawson

Status: Draft

Problem Statement

The Event Listings Page (“Listings”) will be the next Valkyrie Initiative pilot to migrate Listings page rendering out of Core. The Frontend Enablement team will be the primary driver of this work and the Listings team will provide support, so we need to identify what work needs to be done and what work does not.

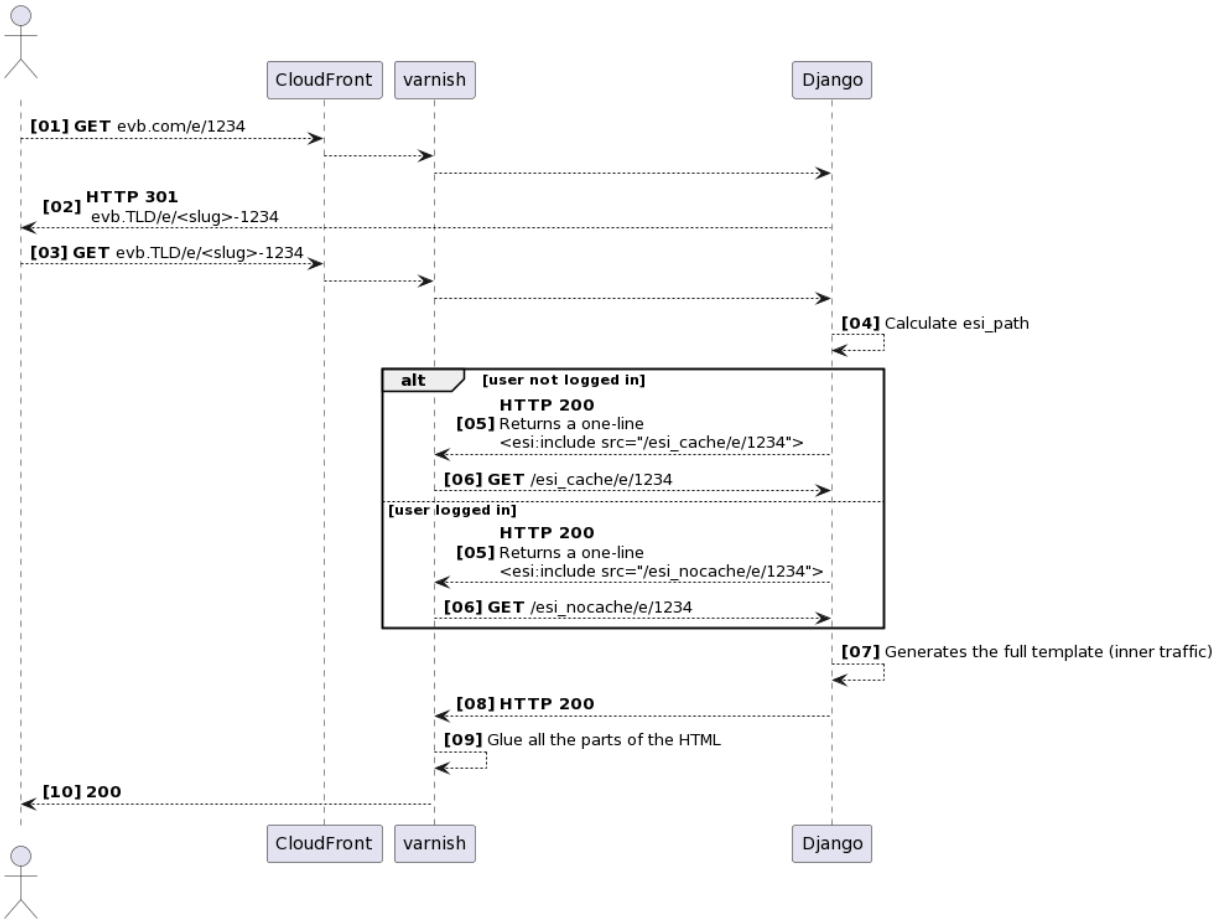
Requirements

“Rendering out of Core” means many things to many people. Following are our requirements for success.

1. Inbound Listings page requests routing directly to Listings Webapp TLZ instead of Varnish/Core
2. Page caching via Cloudfront instead of Varnish
3. Listings “outer traffic” logic in front of Cloudfront cache and no longer in Core Django view
4. Listings EB-UI React apps (`listings`, `listings-protected`) combined into a single, new Next.js React app outside of EB-UI
5. Next.js app server-side rendering in new, feature team-managed Listings Webapp TLZ instead of in Core with Mako templates and React Render Service
6. Gathering Listings context data can remain in “inner traffic” Core Django view, but Listings React app must be able to request this context data of Core from a TLZ for rendering. This includes verifying if requester has permission to view protected Listings pages
7. Maintains functional and UI parity with current Core rendering, except in cases where Listings team deems something obsolete or unnecessary
8. No net new latency or other performance regressions

Current Rendering in Core

Render Flow



Listings outer/inner request traffic flow

(TODO: Add infra/request flow diagram to replace or supplement written steps below)

1. SRE Cloudfront
2. Haproxy
3. Varnish - /e/
4. Core
    a. Routing (production.py)
    b. Dispatch - outer
        i. Core EB DB
        ii. Kafka (affiliate tracking)
        iii. Split
    c. ← Redirects, cookies and/or esi:include tag
5. Varnish
    a. ← Cached page if hit
    b. → Request /esi_ if miss
6. Core Django

    a. Routing (production.py)
    b. Dispatch - inner
    c. View get/post()
    d. View get_context()
        i. Core EB DB
        ii. PySOA
        iii. Listings BFF
    e. React Render Service
    f. ← Mako template rendered page
7. Varnish
    a. Cache page if /esi_cache
8. Haproxy
9. SRE Cloudfront

Routing

After a request passes through managed Cloudfront, Haproxy, and Varnish and arrives in Core, Django (production.py) maps the url path to an ELP dispatch function which is responsible for handling both the inner and outer traffic.

Middlewares

All requests handled by Core Django pass through a series of Django middlewares that quietly provide functionality "for free". Some of this functionality is essential to ELPs, as well as other pages on eventbrite.com, and will need to be replicated in some form. Note that all of these will be applied in the "outer traffic" requests.

Middleware possibly necessary for ELPs, final list TBD:

- Datadog tracing (code)
- Correlation ID (code)
- Page view logging to Kafka (code)
- Force secure cookies (code)
- Session and "secure session" cookies (code)
- Session store in redis (code)
- "New style" analytics actions? (code)
- Page metrics with `pymetrics` (code)
- Set user on request object (code, oauth version)
- Mobile/Eventbrite client identification (code)
- Vanity domains? (code)
- CommonMiddleware - append slash (doc)
- CSRF protection (doc)
- X-Frame-Options clickjacking protection? (code)

- EB+Django security, e.g. `Referer-Policy` header (code, doc)
- Guest ID cookie (code)
- Analytics cookie actions? (code)
- OAuth token login cookie? (code)
- Auth cookie renewal? (code)
- Locale/language (code)
- Waypoints? (code)

Outer traffic

The "outer traffic" code is in the Django dispatch function previously routed to and is run for every ELP request before we look in Varnish for the cached page. Ultimately it either returns an <esi:include> tag that tells Varnish to attempt to serve the page from cache, or it continues on to the "inner traffic" code for a full server-side rendering of the page.

- Notable behavior
    - 4xx HTTP responses
        - No event found
        - Event deleted
        - Spammer event deleted
    - 3xx HTTP redirects
        - To parent event if closed and in a series
        - To event "preferred" TLD if request TLD doesn't match
        - To creator affiliate param if event owner
    - Varnish esi:include tag "safe redirect" to `/notavailable` for canceled/draft event
    - Affiliate tracking
    - Set cookies
        - Affiliate
        - Middlewares
    - Bot detection
- Services
    - Core DB
        - Event
        - Affiliate
        - TeamSetting - if `tid` request param set
    - Cerberus
    - Velocity counter (Redis) - used to force caching during high traffic, possibly no longer needed
    - Kafka (affiliate tracking)
    - `DefaultMetricsRecorder` from `pymetrics` – not sure where it records to
    - Split
    - Datadog

Page caching

We currently cache listing pages in Varnish, but we bypass it for logged in users. The “outer traffic” Django code returns an `esi:include` tag with either an `/esi_cache` or `/esi_nocache` src attribute. This tells Varnish to either check the cache for the page requested or go directly to origin. A request to origin leads to the “inner traffic” code in the same Django dispatch function as “outer”, which ultimately results in a fresh page rendering.

- Notable behavior
    - Sets CSRF cookie if necessary
    - Force caching - to ease load on Core during high traffic

Inner traffic

The “inner traffic” code is in the same Django dispatch function as the “outer” and is only run for `esi_*` requests sent from Varnish. Inner traffic passes through the outer traffic code, but only some of it is run, namely the 4xx response checks, the parent redirect, and Datadog tracing.

Dispatch sends the request to the `ListingLanding` view, which inherits from `BaseEventLanding` where all of the work is done to render the page in its get/post() method.

- Notable behavior
    - 4xx HTTP responses
        - Invalid HTTP request method (only GET/HEAD/POST)
    - Varnish esi:include tag “safe redirect” to `/notavailable` for canceled/draft event
- Services
    - Core DB (Affiliate)
    - Datadog
    - `DefaultMetricsRecorder` from `pymetrics`

Process request (GET, POST params)

The `BaseEventLanding.get()` method handles both GET and POST requests, the latter happening when the requestor submits a password for a protected event listing. Its primary work is to process incoming HTTP params, manage protected event password cookie, and coordinate the other two major bits of work: gathering context data and rendering the page.

Notable Behavior	Services

• Process HTTP params
  ◦ Preview overrides - preview, language, vertical
  ◦ Discount
  ◦ Invite/missive
	• Core DB
	  ◦ Event
	  ◦ Discount
	  ◦ Waitlist
	• Memcache (protected password)

○ Released waitlist • Retrieve protected password - checks a couple params, then uses Guest ID cookie and eid to check memcache for a stored password • Gather context data (see section below) • Cache protected password - from context data • Cancel caching page - from context data • Render page (see section below) • Set cookies - code marked for removal	• Kafka (AnalyticsWriter) • Datadog

- Notable behavior
  - Process HTTP params
    - Preview overrides - preview, language, vertical
    - Discount
    - Invite/missive
    - Released waitlist
  - Retrieve protected password - checks a couple params, then uses Guest ID cookie and eid to check memcache for a stored password
  - Gather context data (see section below)
  - Cache protected password - from context data
  - Cancel caching page - from context data
  - Render page (see section below)
  - Set cookies - code marked for removal
- Services
  - Core DB
    - Event
    - Discount
    - Waitlist
  - Memcache (protected password)
  - Kafka (AnalyticsWriter)
  - Datadog

Gather context data

The vast majority of the code needed to render the ELPs in core is devoted to gathering context data from various sources and transforming it into the shape needed for rendering both the

Listings React apps and the Mako HTML template. Some of this context isn't specifically for the Listings apps and is gathered for every eb-ui React application rendered in core.

IMPORTANT: The Listings team is actively moving this get-context functionality out core into a separate Listings BFF service written in Kotlin, with the eventual goal of getting context entirely from the BFF with this section 100% out of core.. Listings team is doing the migration incrementally by service dependency (e.g. Structured Content) and will continue this work into 2024.

- Notable behavior
    - Data requested from many sources
    - Models for transforming data for rendering - a LOT going on here
    - Password/invite check for protected events
    - Determines Mako template and eb-ui app to render - listing or listing-protected
    - SEO json+ld
- Services
    - Core DB
        - Affiliate
        - EventFormats
        - EventCategories
    - PySOA
        - Cerberus
        - Event
        - EB API (tracking beacon) - Is this actually Legacy SOA?
        - Ticket availability
        - Organizer
        - Venue
        - Geo
        - Payment capability
        - Refund
        - Destination
        - Image
    - Listings BFF
    - Redis (waiting room queue w/crypto)
    - Split
    - Datadog
    - `DefaultMetricsRecorder` from `pymetrics`

Render page (React and Mako)

All eb-ui pages are rendered in two phases in core and Listings is no exception:

1. Server-side render eb-ui app via React Render Service
2. Render Mako template with RRS output injected into it

This generates the complete HTML page that gets sent back to the requester courtesy of Django's `render_to_response` function.

There are two eb-ui apps and two Mako templates, `listings` and `listings-protected`, and `listing.html` and `listing_protected.html` respectively.

- Services
    - React Render Service
    - Datadog

HTML templates

These are Mako templates that Django uses to assemble the page HTML to send back to the requester.

listing.html

listing_protected.html

React apps

There are two eb-ui React apps needed to render all types of Event Listings:

1. `listings` - for public events and protected events if the requester has permission to view it, either by invite or correct password
2. `listings-protected` - for protected events the requester hasn't yet proven they have permission to view, with optional password form

NOTE: There's a third eb-ui app, `listings-preview`, that's used to show a preview rendering of a draft event to the creator before publishing. It's extremely simple (displays the preview in an iframe) and renders in core via a separate Django view, `EventListingPreviewView`, also extremely simple. TBD if it should also be part of the Valkyrie migration.

Each app consists of three things: app code, 1st party dependencies (`@eventbrite` packages) and 3rd party dependencies.

`listings`

The much larger of the two apps, it contains everything needed to render an ELP the requester has permission to view. It also optionally renders a ticket selection widget (owned by Checkout team) in an iframe in the page.

- App stats
    - 100% Typescript
    - 127 component files
    - 61 SCSS files
    - BEM-style CSS

    - eds-* css overrides: 97 instances, 29 files
    - eds-* className prop: 75 instances, 28 files
- 1st party dependencies (including transitive)
    - Total: 138
    - Direct: 69
    - eds: 65
    - infra: 42
    - attendee: 19
    - organizer: 8
    - tools: 4

`listings-protected`

A simple app that renders if a requester doesn't yet have permission to view the listing. If the event is password protected, this app displays a password form.

- App stats
    - 100% Typescript
    - 13 component files
    - 8 SCSS files
    - eds-* css overrides: 5 instances, 2 files
    - eds-* className prop: 3 instances, 2 files
- 1st party dependencies (including transitive)
    - Total: 117
    - Direct: 18
    - eds: 62
    - infra: 32
    - attendee: 13
    - organizer: 7
    - tools: 3

Proposal

Overview

At a high level, the Valkyrie initiative aims to do a handful of things:

1. Enable rendering out of Core
2. Enhance DX
3. Improve web performance
4. Support team autonomy, in part by

To do this, we're working to identify and adopt industry standard tooling and best practices, develop ready-made solutions for common problems, and decentralize web frontend development and deployment.

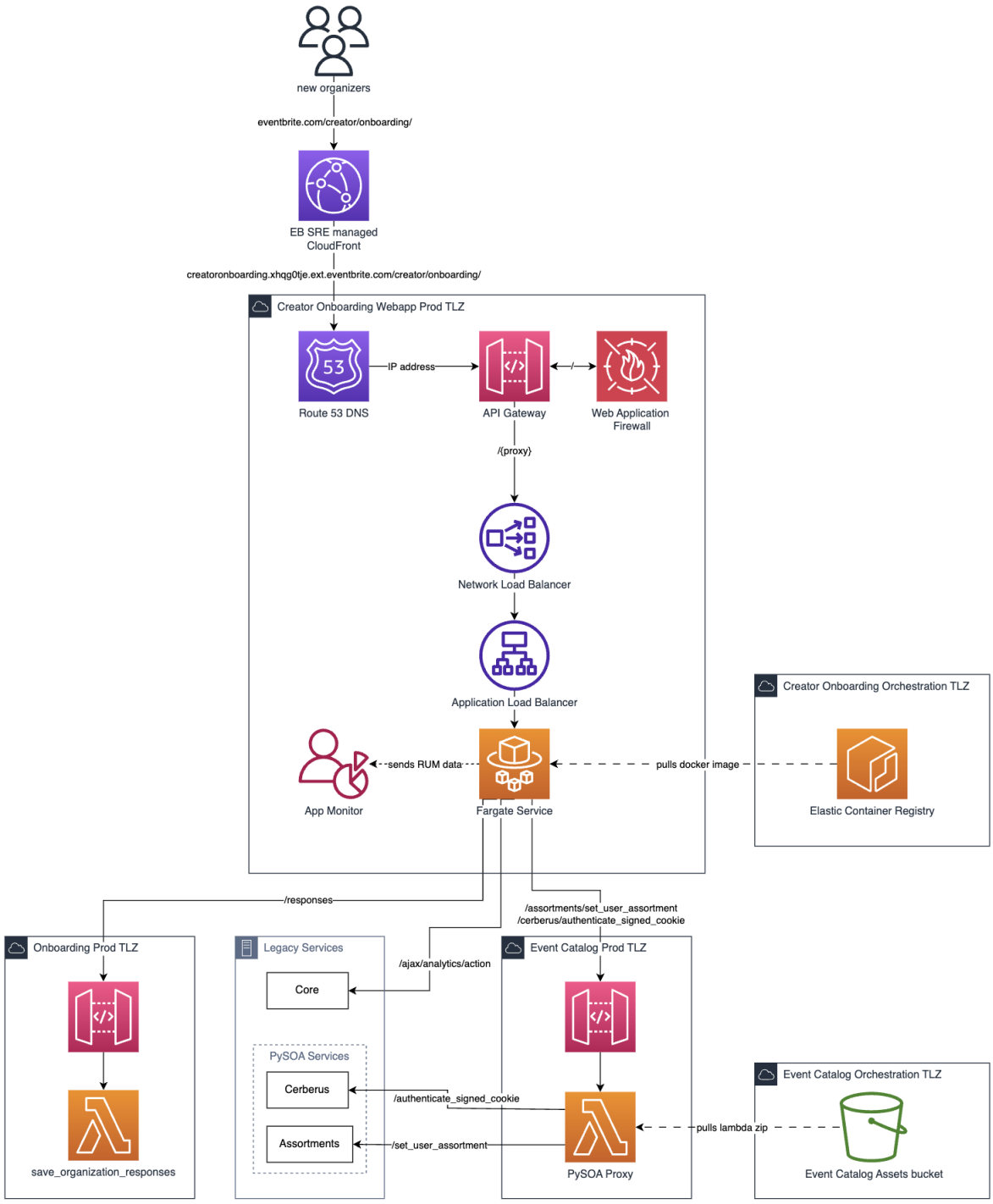
In practical terms, this means Next.js React applications that unify both client and BFF code into a single development experience, support server-side rendering, and offer a number of performance optimizations out of the box. They'll be built in independent repos, deployed to team-managed TLZs, and supported by improved shared libraries migrated from eb-ui and new Valkyrie libraries for common functionality like cookie management and CSRF protection.

As such, we propose the following for the Valkyrie Event Listing Page migration project:

- `listings` and `listings-protected` eb-ui apps and Mako templates migrated into a single Next.js app
- Shared library/component dependencies migrated out of eb-ui into new repo(s) for general reuse with some targeted refactorings (scoped CSS) and upgrades (3rd party deps)
- New Next.js app repo with its own CI/CD/tooling, and infra in new Listings Webapp TLZ
- Routing via SRE Cloudfront to Listings TLZ
- Page cache Cloudfront distro in Listings TLZ with A/B experimentation @edge
- Outer traffic logic migrated to Typescript/Node.js Lambda@Edge
- Request handling and server-side rendering via Next server runtime
- Key middleware capabilities recreated as services/libraries/Next.js middlewares, including:
    - Session, guest, and correlation cookies
    - User authentication from cookie
    - CSRF protection

What we think should NOT be included in this work:

- Context data gathering will NOT be migrated to the Next.js app, this will continue to be handled in core while Listings team continues building out their existing Listings BFF

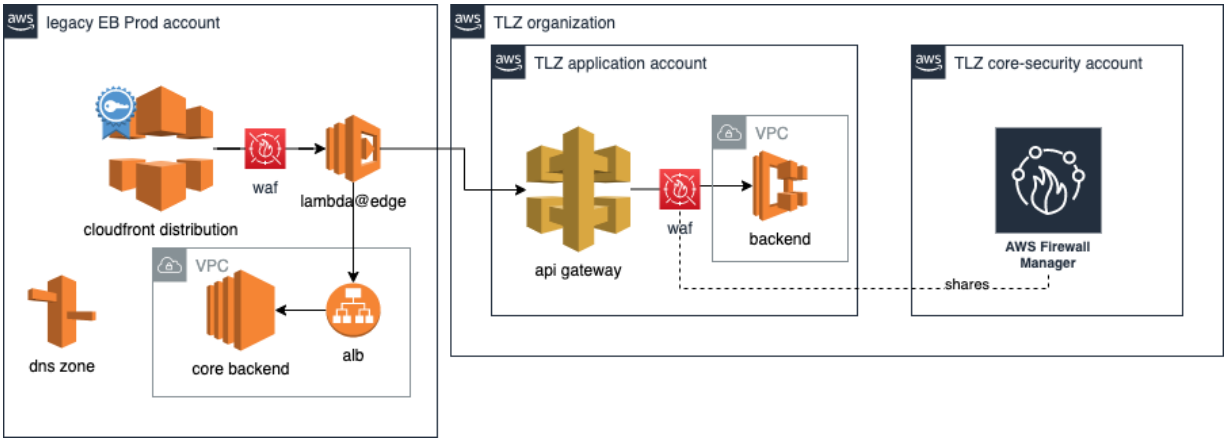


From [creator-onboarding-webapp repo](#) (TODO: replace w/Listings proposed infra)

Routing

Routing will be handled by the SRE managed Cloudfront distribution, which will route /e/* requests to the Listings Webapp TLZ.

The Listings TLZ will need its own WAF, but there's an Edge-as-a-Product Proposal (Path 1) that would result in a Security-provided WAF via AWS Firewall Manager, once implemented.



From Edge-as-a-Product Proposal (Path 1)

To support prod testing and a cautious launch of the Listings Next.js app, we'll need a Split.io feature flag and support from SRE to route inbound requests to Core or the Next app based on the FF (TODO: How does this work? L@E?)

Dependencies:

- SRE - Legacy Cloudfront routing change
- SRE - Temporary Split-based routing
- SRE - Edge-as-a-Production Path 1 implementation (TODO: requirement? nice-to-have?)

Page caching

Event Listing pages will be cached in a new Cloudfront distribution in the team-managed Listings Webapp TLZ. This gives the Listings team full control of their caching policies and any additional functionality they need to attach in the form of Lambd@Edge or Cloudfront Functions.

Because the ELPs are served via the eventbrite.com (+other TLDs) domain, Path 2 of the EaaP proposal (route directly to TLZ via DNS) is not an option to us. This means all ELP traffic will minimally go through 2 WAFs and 2 Cloudfronts, but there's no real way around that. Managing a

2nd CF in the Listings TLZ is necessary to deploy the “outer traffic” logic as a Lambda@Edge AND give the Listings team the autonomy to release updates to it independently.

Dependencies:

- SRE - Edge-as-a-Production Path 1 implementation (TODO: requirement? nice-to-have?)

Outer traffic

This functionality can be moved into a **Lambda@Edge** in front of the page cache. It makes requests to several services and so cannot be run in a Cloudfront Function.

Naturally these service calls will add to response times even when the page is a cache hit. There’s an opportunity to move some of this functionality to the “inner traffic” code (i.e. Next.js app), for instance most of the 3xx and 4xx responses could be cached in CF, though it’s unclear what our requirements are for reflecting changes to the event, like when it’s deleted. However, it appears some of this logic will have to stay “outer”, like affiliate tracking/redirect, and TLD redirect, both which depend on data from Core DB.

Data access:

- Core DB - `Event`, `Affiliate` and `TeamSetting` tables
- Cerberus - Existing PySOA proxy
- Split - There are currently two Split feature flags which have to be retrieved from Split’s servers and from Node.js, which doesn’t have access to our split-synchronizer optimization. TODO: Do we need these splits? Do we need support for low latency Split FFs in the future?
- A/B experiments - Not a current need, but Listings team wants to eventually run A/B tests @edge
- Kafka - Affiliate tracking. TODO: Is there retry logic? Do we need to wait for a response?
- Datadog - We’re adding traces to an existing DD span, but I’m not sure when it gets sent. TODO: Do we need DD traces for the entire request/response cycle? Or other?
- `DefaultMetricsRecorder` from `pymetrics` - TODO: Where is it going? Is it necessary?

Dependencies:

- Access to specific Core DB tables/fields via new API/service
- A/B testing @edge, Statsig or otherwise
- Kafka access
- Cookie management - session, guest and correlation, minimally
- CSRF protection

Risks:

- Performance - If/when we go multi-region, some of our service requests (e.g. Core DB) might become latency issues until those services are also multi-region.

- DX - This might technically be something we could put in one or more Next.js middleware functions, but deploying them as L@E functions might be too difficult outside of Vercel. For instance SST is apparently not supporting this. If it has to be separate L@E code then we'll need to figure out how to incorporate it into the dev server.

Next.js App

With one exception, the remaining parts of the Core ELP rendering flow will be handled by a single Next.js application in the Listings Webapp TLZ. It will consist of:

- React client code
- HTML templates as React components
- GET/POST request processing code
- Server-side rendering

React apps

The two eb-ui ELP React apps, `listings` and `listings-protected`, will be refactored into a single Next.js application in a new git repo, completely decoupled from eb-ui. Code splitting techniques will be used to minimize the JS needed to render the normal and "protected" views.

App migration:

In order to use the latest version of Next, we have to upgrade a number of key 3rd party dependencies, including React to version 18, two major versions ahead of what's currently used in eb-ui.

We're also migrating Valkyrie components to scoped CSS, specifically CSS Modules using SCSS. This means we can include only the CSS needed in our app builds (no more 500 KB eds.css on every page), and we can eliminate a whole class of style-related regressions by disallowing implicit dependencies between arbitrary components via global style overrides. CSS encapsulation is essential as we move toward decentralized web frontend development and ultimately implement micro-frontends.

In all we'll apply upgrades and CSS modules to the 140 components and 69 SCSS files between the two eb-ui apps.

Refactoring will also include adopting Next-native components, like next/dynamic for lazy loading (instead of `loadable`) and next/image for optimized images. Not all Next optimizations are necessary for this migration and can be left for later improvements.

Shared components:

Virtually all existing eb-ui apps depend on a significant number of shared libraries/components in eb-ui, of which there are over 400, and the Listings apps are no exception. The dependencies can be direct, or transitive, when an `@eventbrite` dep has its own `@eventbrite` dep. eb-ui has a

problematic web of interdependencies between its packages, which results in an abnormally large 1st party dependency tree for eb-ui apps.

To illustrate: `listings` has 68 direct, but 138 total. However, `listings-protected` only has 18 direct and 117 total! `listings` only has 22 deps the other, much smaller app does not. On the plus side, there's a tremendous amount of overlap between them when it comes to migrating those library packages out of eb-ui as a part of what we're calling the Shared Components problem..

For a number of reasons, it's not easy or advisable to try to use packages in eb-ui outside of eb-ui, and it's very difficult to upgrade some of the 3rd party deps in eb-ui because of the way it's set up with lockstep versioning. Meaning, every package in the monorepo needs to be on the same version of every shared dependency, so upgrading a dep that requires any amount of work or refactoring becomes prohibitively expensive in time, such as React 16x -> 18x. This also means that refactoring components to CSS Modules in-place in eb-ui would require large all-at-once changes throughout the monorepo.

Valkyrie apps are in independent repos, but they continue to depend on shared libs in eb-ui, so part of any Valkyrie migration is copying all shared packages the app depends on out of eb-ui into a new shared space (one or more new repos) where we can apply our upgrades and scoped CSS. However, in order to avoid maintaining two copies of each shared lib (one in eb-ui and one out) for potentially years while apps are migrated out of eb-ui, we're developing a migration strategy that will make these upgrades backwards compatible and/or refactor dependents in eb-ui to adapt to these shared lib changes so every consumer is using the same shared packages.

As Listings is the first Valkyrie migration with shared lib dependencies, we're budgeting time to migrate and update the shared libs Listings needs. This will include setting up the shared space and associated tooling/CI/CD for publishing the packages, as well as means to support polyrepo development and workflows.

Dependencies:

- Detailed Shared Components strategy

HTML templates

The Mako templates in Core for rendering the Event Listings page HTML will be rewritten as a single `Document` React component that Next.js will use during SSR.

It will be equivalent to the HTML Core produces today, except in cases where bits of HTML are deemed obsolete. An audit of the Mako templates should precede this work.

Process request

A custom Next.js getServerSideProps() function can handle both GET and POST Event Listing page requests, and will effectively replicate the `BaseEventListing.get()` handler in Core:

- Accept and validate GET params for previewing, discount, invite/missive, and released waitlist

- Retrieving password from POST data and/or password cache - TODO: Core stores successful password submissions in memcache so user doesn't have to resubmit for the same event page. How should we handle this?
- Saving password in cache
- Requesting context data - See "Gather context data" section below

POST requests are for password form submissions on protected events. This POST full page turn can eventually be refactored to be an async API request from the client if we want, but that should wait until the Listings BFF work is complete.

We also have the option to change our mechanism for password protecting events, but we should consider limiting the scope of this project where we can. Either way, how we cache accepted passwords will need to be updated.

Data access:

- Core DB - `Event`, `Discount` and `Waitlist` tables
- Kafka - logs EventViewAction which writes to Kafka via AnalyticsWriter
- Datadog - Adding traces to an existing DD span

Dependencies:

- Access to specific Core DB tables/fields via new API/service
- Kafka access
- CSRF protection
- Means to save/cache a user's accepted passwords , or rework of password protection mechanism

Gather context data

IMPORTANT: We don't think we should rewrite the context gathering functionality currently in `BaseEventLanding.get_context()` as part of this Valkyrie migration project.

It accesses over a dozen services, but more significantly there's a tremendous amount of work done to munge the data into a form the React app expects. On its own this wouldn't be a showstopper, but the Listings team is already working on moving all of this functionality into the Listings BFF service, work that will continue in 2024.

As such, any work that we would do to rewrite `get_context()` as Node.js code in the Next app would ultimately be throwaway effort.

Instead, we'd like to temporarily expose the output of `get_context()` either via a new Core APIv3 endpoint OR by modifying the existing `BaseEventLanding.get()` handler to detect a custom header and respond only with the context JSON, similar to how almost all other React views in Core accept `X-EB-App-Context`.

The Next.js app would request this context data from Core during SSR and then simply switch to request from the BFF once it's complete.

Dependencies:

- Listings - Help safely make this Core change to support requesting ELP context data.

Render page

There are a few out-of-the-box ways of rendering individual pages in a Next.js app: client-side, server-side, and statically generated. Simply exporting a `getServerSideProps` function from the page component file will signal to Next that it should be SSRed.

Middlewares/Capabilities

Cookie management

TODO: Describe part of this problem we'll need to solve, namely setting/handling identity cookies - session, guest, correlation.

CSRF protection

TODO: What we've already done and changes that might be necessary to support APIv3 calls from components owned by other teams, like Ads. We probably don't want to have to write our own BFF API endpoints for each of these, but in order to support calls to existing endpoints we'll need a CSRF cookie that Core understands. Etc.

A/B experimentation @edge

TODO: Brief description of problem and links to related docs.

Split.io feature flags in Node

TODO: Work to address problems accessing Split FF from Node in a performant way.

Legacy metrics/analytics

TODO: Kafka, etc

Datadog tracing or similar

TODO: We have end-to-end Datadog traces of the entire request/render cycle in Core, with a number of spans. Can/should we support something similar, and across service boundaries?

Protected listings password caching

TODO: Problem is described briefly in the "Process request" section above, but it's unclear how best to handle this.

Rollout

TODO: Quality, scaling, load testing, requirements for eventual rollout of Next.js app, means for doing that safely. Doesn't have to be detailed, but I think should be mentioned.

Impact

TODO: How the proposal will positively and/or negatively affect Eventbrite (pros and cons), including engineering process, development productivity, site performance, storage and user experience.

Migration

TODO: Break down into discrete problems and delivery units. Suggest some sequencing.

Alternatives

These aren't alternatives as much as they are minor variations on this proposal.

Include context data gathering in Valkyrie migration

We could rewrite all the `BaseEventLanding.get_context()` context data fetching and munging code in Typescript/Node.js for the Next.js app. This would collocate the BFF and client code, significantly improve the DX experience of working on both, and further the Valkyrie BFF patterns and solutions, a key objective of this early pilot migration work.

However the Listings team is currently working on rewriting this exact functionality in their own, already in-use Kotlin Listings BFF service, which would ultimately make any Valkyrie Listings context gathering migration work redundant and disposable.

Don't include GET/POST request handling in Valkyrie migration

One way of making the ELP context data available via HTTP requests to Core could be to add support to the `BaseEventLanding` Django view for the `X-EB-App-Context` header. Virtually every other React view in Core recognizes this custom header and will return just the context data in JSON form, instead of the rendered HTML for the view.

To do this simply, `X-EB` requests would flow through the same `BaseEventLanding.get/post()` handlers, which will process/validate params, etc. This means we wouldn't need to replicate this behavior in the Next.js app. But this also means that's even more code that remains in the Core Listings rendering path, and this is functionality that should live with the React app.

IGNORE - TO DELETE 👇

Abstract

As a Valkyrie pilot project, we propose migrating the Event Listing Page React apps (`listings` and `listings-protected`) out of eb-ui into a new Next.js app in its own git repository, migrate its pre-Varnish ("outer view") logic, HTML page template, and server-side rendering out of Core into a team-managed TLZ, and finally page caching out of Varnish into Cloudfront. We're also proposing that gathering the context data necessary to render the React app should continue to happen in the existing Core Django view for now, but should be accessible from the Next.js app via an HTTP request during server-side rendering.

Background

The Frontend Enablement team has been evaluating the Event Listing Page (ELP) web view as a possible high-value candidate as a key 2024 Valkyrie pilot migration project. Valkyrie is a Platform initiative to support teams rendering their web Frontend React apps entirely out of Core by leveraging industry standard frameworks like Next.js, supporting new patterns like Backend For Frontends (BFFs) and embracing team autonomy by both decentralizing Web Frontend development and providing ready-to-use infra/CI/CD tools, configurations and constructs that support hosting independent React apps in team-managed AWS TLZs.

The FE Enablement team is in the final stages of our first app migration (Creator Onboarding) and during a 2024 planning review with T-Staff the team was encouraged to pursue high-value, Consumer-facing features for our next pilot. Event Listing Page was quickly identified as a top candidate due to its incredibly high amounts of traffic, its immense SEO value, and its critical role in the funnel to Checkout.

We have taken some weeks to analyze the existing eb-ui apps, the Django view logic (inner and outer), the data sources needed to gather the needed context data for rendering, and how password protected pages are handled. After consulting extensively with the Listings team, the FE Enablement team is convinced ELPs should be our next Valkyrie pilot migration.

Proposal Overview

At a high level, FE Enablement will merge the `listings` and `listings-protected` eb-ui apps into one new Next.js application that will be responsible for rendering all Event Listings Pages and the not-yet-authorized UI for protected events. We will also migrate the shared components the Listings React apps needs out of eb-ui into one or more new git repos where we'll apply 3rd party

dependency upgrades and refactor the shared components so they're usable in the Next.js app as well as existing eb-ui dependent packages.

The Next.js app code will be in a new git repo, with its own CI/CD pipeline, and it will be deployed to and hosted in a new Listings Webapp Prod TLZ, which the SRE managed Cloudfront will route /e/* requests to.

The Listings Mako templates in Core used to generate the HTML response to ELP requests will be converted into one or more Document components in the Next.js app. The Listings "outer" Django view that sits in front of the Varnish page cache will be rewritten in Typescript and likely put in a Lambda@Edge in front of a team-managed Cloudfront distribution in the new TLZ. This Cloudfront will replace the Varnish Listings page cache.

The Django view get/post methods will be translated into Next.js code that's run when server-side rendering. However, the Django view `get_context()` code will NOT be translated into an equivalent BFF in Next.js. The Listings team is already in the process of moving all of that get_context logic and related service calls into a separate Listings BFF written in Kotlin. As that work is ongoing, any BFF code we'd add to the Next.js app would effectively be extremely time-costly throw-away work. In this proposed scenario the Listings team would work with us to do a little refactoring of the Django view and expose a new Core API endpoint that returns all the necessary Event Listing Page context data for an ELP page request, instead of the fully rendered HTML as it does now. Then the Next.js app will just make this single call to get the context data for server-side rendering. This also means password checks for protected listings will likely continue to happen in the Django view code.

FE Enablement will also be responsible for developing solutions to a number of general Valkyrie subproblems that arise from moving rendering out of Core/Varnish. This includes things like Statsig-like A/B testing @edge, cookie management, and SEO metadata.