

Micro Frontend Architecture

Building an Extensible UI Platform

Erik Grijzen



At New Relic, we used a **micro frontend architecture to migrate from several monolithic products applications to one single platform where our users have one unified experience**. This **new platform UI is broken down into separate micro applications, views, and other extension points**. This extensible way of building a platform allows us to reduce the toil for our development teams to work on new features and bring value to our customers much faster. This new architecture was not only built for internal development but also allows third-party (customers) developers to extend our platform for their own special use cases. Based on our real-world experience going through this transition, we will take a look at the benefits that come with such an architecture, such as ease of deployments, increased team autonomy, localized complexity, and more.

There were also many challenges along the way and we will reflect on how we tackled some of the hardest problems we faced regarding performance, resiliency, UI consistency, scalability, and even organizational changes. I'm a Lead Software Engineer with a focus on front-end technologies building platform UI @ New Relic. Web performance enthusiast and clean code lover.

Safe Harbor

This presentation and the information herein (including any information that may be incorporated by reference) is provided for informational purposes only and should not be construed as an offer, commitment, promise or obligation on behalf of New Relic, Inc. ("New Relic") to sell securities or deliver any product, material, code, functionality, or other feature. Any information provided hereby is proprietary to New Relic and may not be replicated or disclosed without New Relic's express written permission.

Such information may contain forward-looking statements within the meaning of federal securities laws. Any statement that is not a historical fact or refers to expectations, projections, future plans, objectives, estimates, goals, or other characterizations of future events is a forward-looking statement. These forward-looking statements can often be identified as such because the context of the statement will include words such as "believes," "anticipates," "expects" or words of similar import.

Actual results may differ materially from those expressed in these forward-looking statements, which speak only as of the date hereof, and are subject to change at any time without notice. Existing and prospective investors, customers and other third parties transacting business with New Relic are cautioned not to place undue reliance on this forward-looking information. The achievement or success of the matters covered by such forward-looking statements are based on New Relic's current assumptions, expectations, and beliefs and are subject to substantial risks, uncertainties, assumptions, and changes in circumstances that may cause the actual results, performance, or achievements to differ materially from those expressed or implied in any forward-looking statement. Further information on factors that could affect such forward-looking statements is included in the filings New Relic makes with the SEC from time to time. Copies of these documents may be obtained by visiting New Relic's Investor Relations website at ir.newrelic.com or the SEC's website at www.sec.gov.

New Relic assumes no obligation and does not intend to update these forward-looking statements, except as required by law. New Relic makes no warranties, expressed or implied, in this presentation or otherwise, with respect to the information provided.



Why?

Why did we adopt a micro frontend architecture?



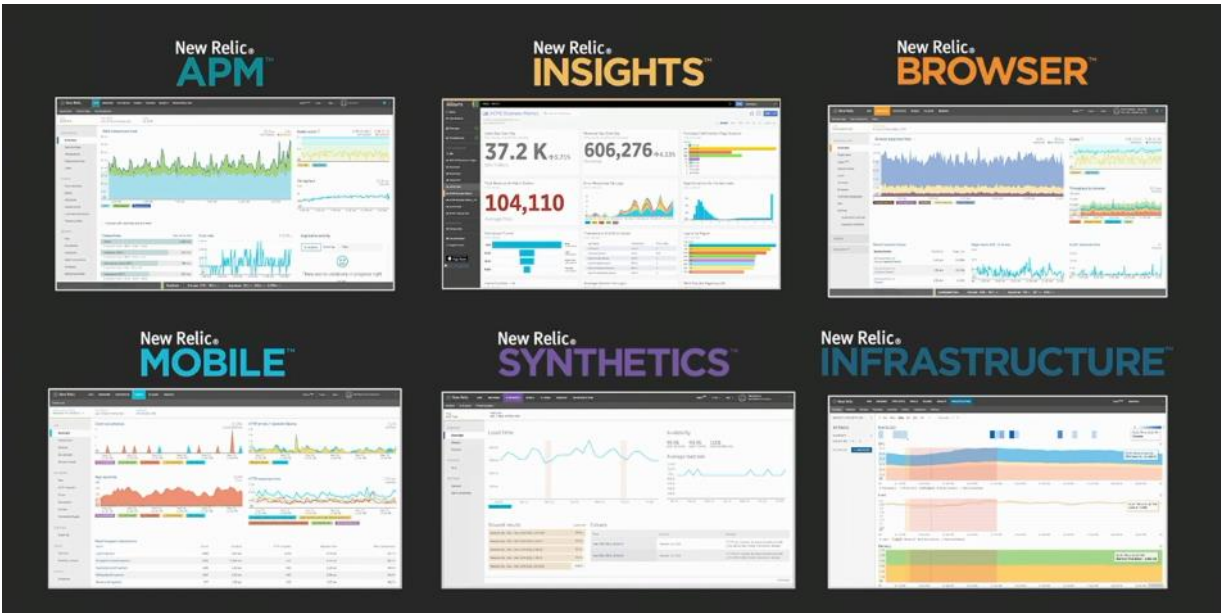
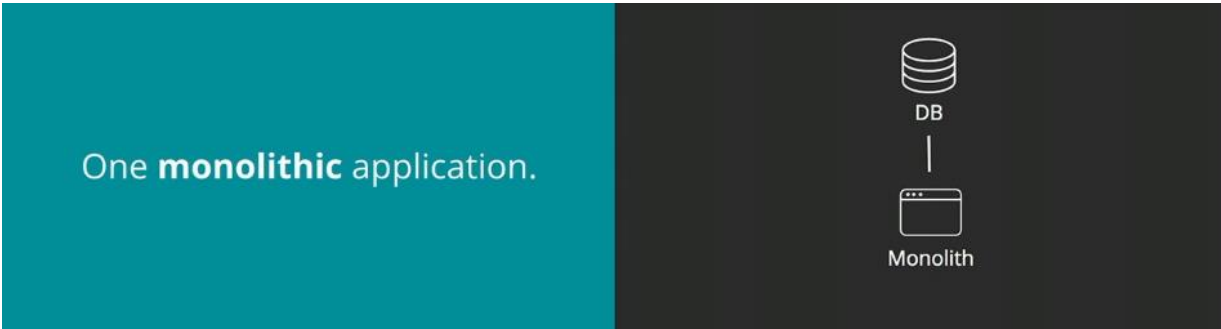
What?

What are micro frontends?
What architecture decisions have we made?



How?

How did we implement this architecture?





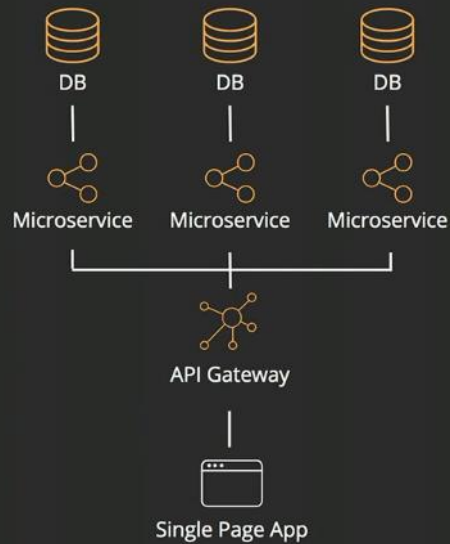
Typical **monolithic** problems.

We started having dependency issues between the products

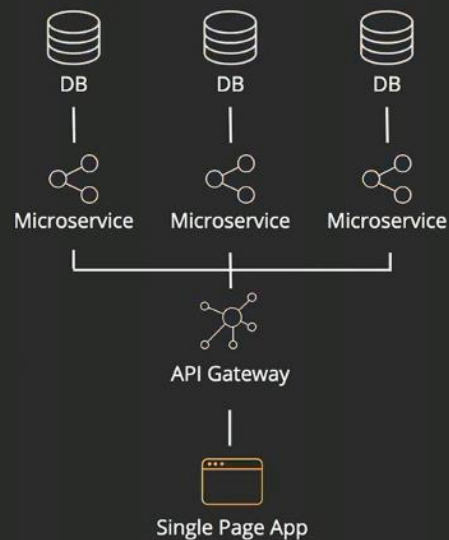


Adopting **microservices**.

Dismantling the backend into **microservices**.




Every product became a **Single Page App**.



On the FE, each product became its own SPA with links at the top that go to each different SPA


More or less **2 years ago...**

We couldn't scale our UI anymore because of 3 main problems




UX Consistency

A more connected and unified user experience.



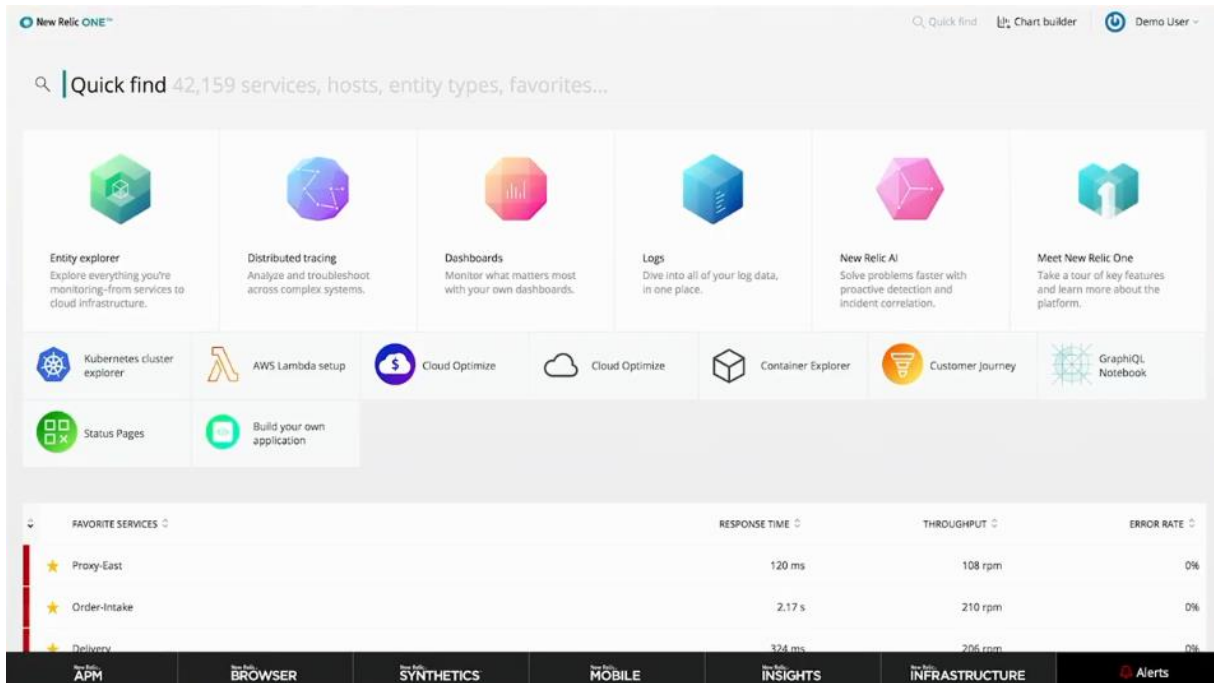
Extensibility

Allow customers to extend the products.



Remove boilerplate

Avoid reinventing the wheels and focus on product innovation.



The screenshot shows the New Relic ONE dashboard interface. At the top, there's a search bar with the text "Quick find 42,159 services, hosts, entity types, favorites...". Below this, there are several tiles for different features: Entity explorer, Distributed tracing, Dashboards, Logs, New Relic AI, and Meet New Relic One. Further down, there are more tiles for specific services like Kubernetes cluster explorer, AWS Lambda setup, Cloud Optimize, Container Explorer, Customer Journey, GraphQL Notebook, Status Pages, and Build your own application. At the bottom, there's a table showing performance metrics for various services.

FAVORITE SERVICES	RESPONSE TIME	THROUGHPUT	ERROR RATE
Proxy-East	120 ms	108 rpm	0%
Order-Intake	2.17 s	210 rpm	0%
Delivery	324 ms	206 rpm	0%

The bottom navigation bar includes links for APM, BROWSER, SYNTHETICS, MOBILE, INSIGHTS, INFRASTRUCTURE, and Alerts.

We started a new initiative to join all our separate products into one single platform called New Relic One.

The New Relic One platform **goals**.

- Easier and faster feature development.
- A more unified UI / UX.
- Improved performance.
- Third party extensibility.

How can we **scale** our **UI development**?



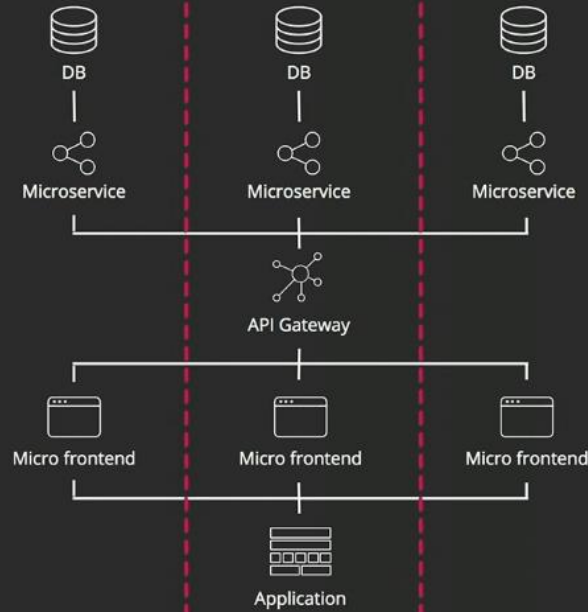
Adopting a micro frontend architecture is our attempt to solve these issues arising from going to one unified platform



The idea behind Micro Frontends is to think about a website or web app as **a composition of features** which are owned by **independent teams**. Each team has a **distinct area of business** or **mission** it cares about and specialises in. A team is **cross functional** and develops its features **end-to-end**, from database to user interface.



www.micro-frontends.org



Bringing the concepts of
microservices to **UI development**.



Easier and **faster**
feature development.

Teams should be
autonomous.

- Independent deployments.
- Cross functional teams.
- End-to-end implementation.
- Full ownership & responsibility.
- Allows parallel development.

Micro frontends should be **small** and **decoupled**.

- Loosely coupled.
- Clear contracts.
- Easier to reason about.
- Easier to add, change, remove.
- Easier to test.

Micro frontends should be **business domain** centric.

- Aligned with business structure.
- Highly specialized teams.

Removing **boilerplate**.

- Creating clear tech standards:
 - Provide good defaults.
 - Easier to switch teams.
 - Innovate on platform level.
- Automating everything:
 - Project creation.
 - Build setup.
 - Continuous integration.
 - Continuous deployment.



A more **connected** and **unified** user experience.

Micro frontends should have a **consistent UI / UX**.

- Design system.
 - Component library.
- Declarative platform APIs.



Improved **performance**.

Micro frontends should be **performant**.

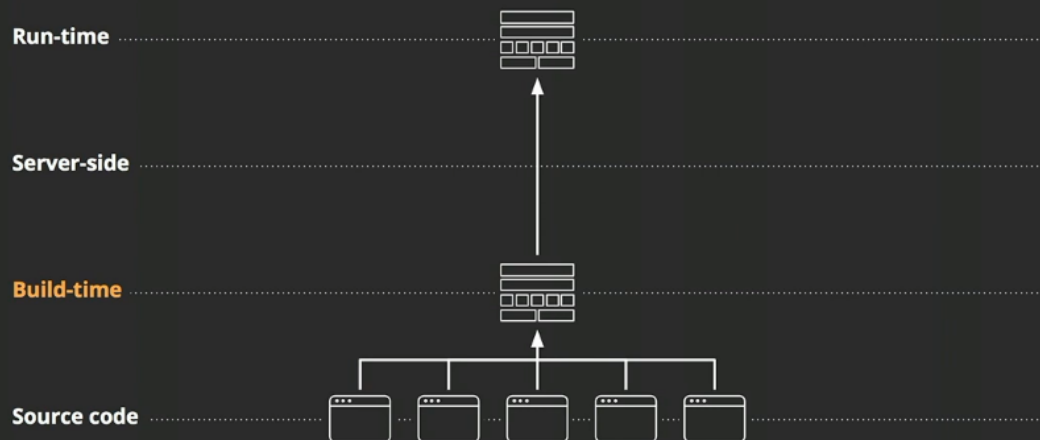
- Shared common dependencies.
- Deduplicate dependencies as much as possible.



Micro frontends **composition**.

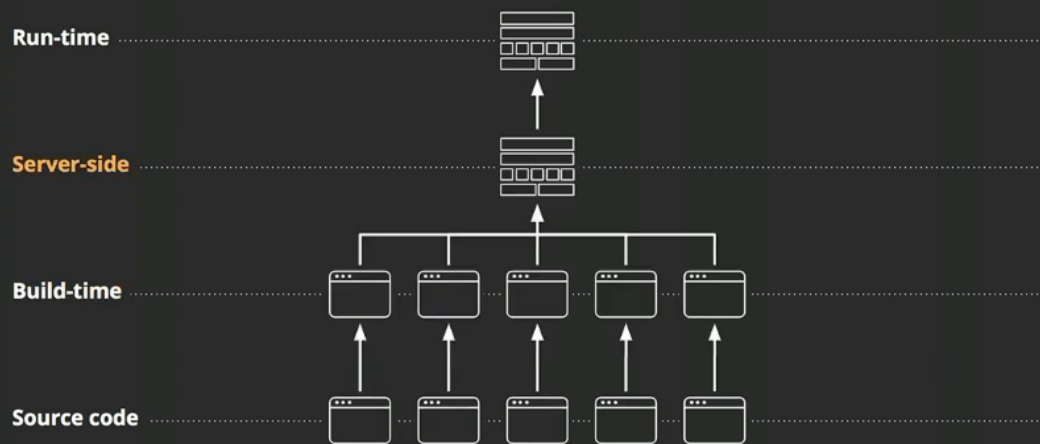
There are 3 ways to achieve this composition

Build-time composition.



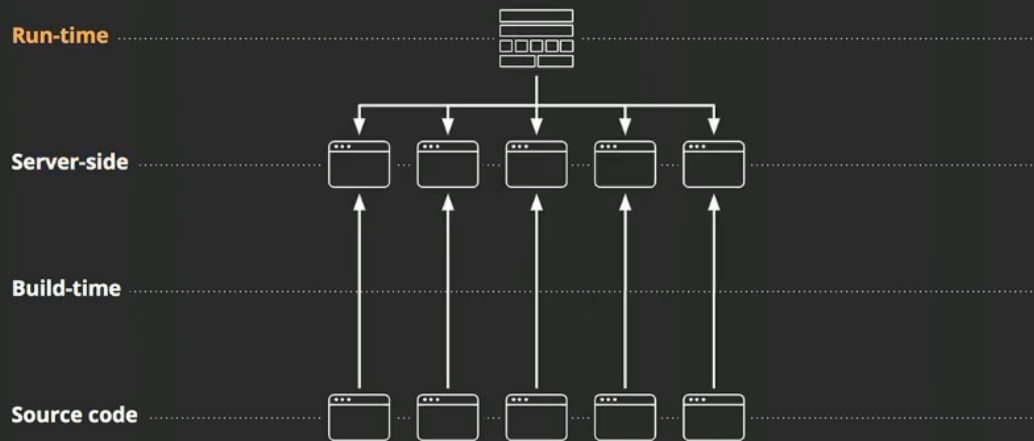
This is where your micro-frontends all live in a monorepo and they get composed together into a single application during build time into a bundle that gets served to the browser.

Server-side composition.



This is best for sites that need to be indexable like e-commerce websites that display differently based on the product page or route.

Run-time composition.



This uses a larger container application that gets loaded, micro-frontends are then lazy-loaded at runtime to compose the UI based on the specific route the larger application is one. We choose this approach because our main app is behind a login page that allows us to do lazy-loading more efficiently after the initial page load.



Our implementation

Extending the platform with the New Relic One CLI.

```
> test-nerdpack@0.1.0 start /Users/egrijzen/Projects/nr3/test-nerdpack
> nr1 nerdpack:serve

Nerdpack:

package.json:
  ✓ name test-nerdpack
  ✓ id 8ba28fe3-5365-4f7f-8f6a-4b8c6c39d8a6

Found and loaded 2 artifacts on test-nerdpack (8ba28fe3-5365-4f7f-8f6a-4b8c6c
Artifacts:

Launchers:
  ✓ my-launcher launchers/my-launcher/nr1.json

Nerdlets:
  ✓ my-nerdlet nerdlets/my-nerdlet/nr1.json

🔧 Built artifact files for:
  * 8ba28fe3-5365-4f7f-8f6a-4b8c6c39d8a6--my-nerdlet built ✓
  * 8ba28fe3-5365-4f7f-8f6a-4b8c6c39d8a6--styles built ✓

✓ Nerdpack built successfully!
★ Starting as orchestrator...

✓ Server ready! Test it at: https://one.newrelic.com/?nerdpacks=local
➔ Server will reload automatically if you modify any file!
```

To extend our platform, we built a New relic One CLI tool that allows us to extend the platform.

```
> nr1 create --type nerdpack --name my-nerdpack
```

You can use the CLI and its create command to create a new repository/nerdpack for your new micro-frontend

```
> nr1 create --type nerdpack --name my-nerdpack
> cd my-nerdpack
```



Nerdpack



package.json

Nerdpack **repository.**

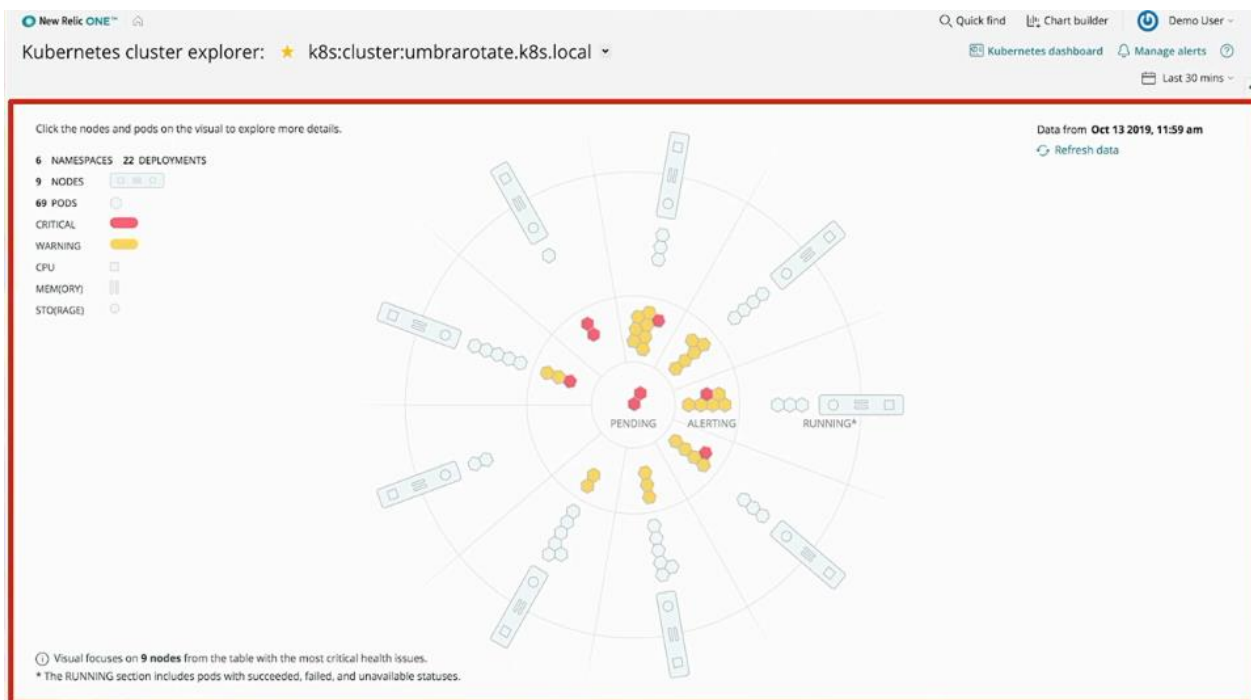
- Project structure.
- Build setup.
- Deployment pipeline.

```
> nr1 create --type nerdpack --name my-nerdpack
> cd my-nerdpack
> nr1 create --type nerdlet --name hello-world
```

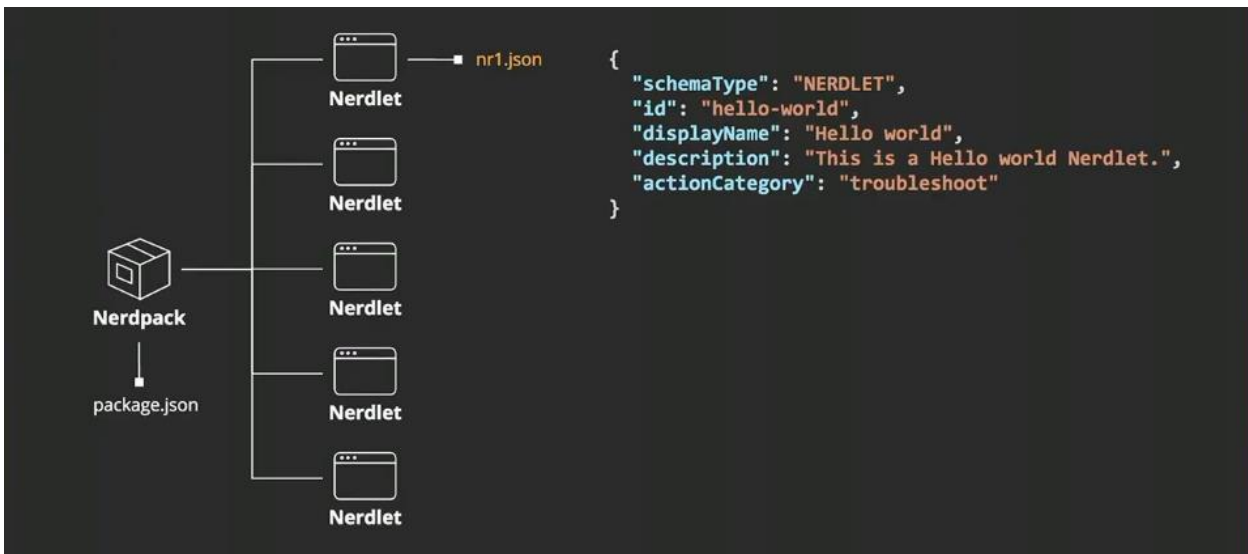
You can now start adding micro-frontends to your repository



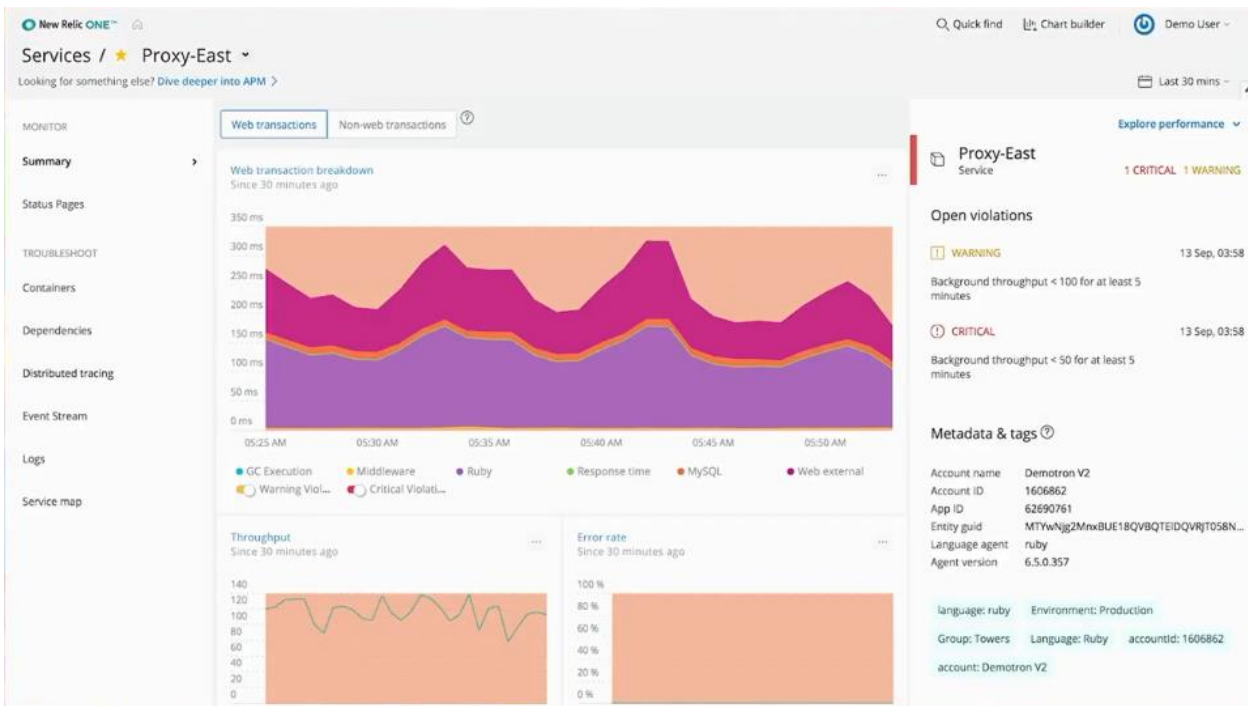
You can have more than one micro-frontends (for a business domain functionality) in the same repository



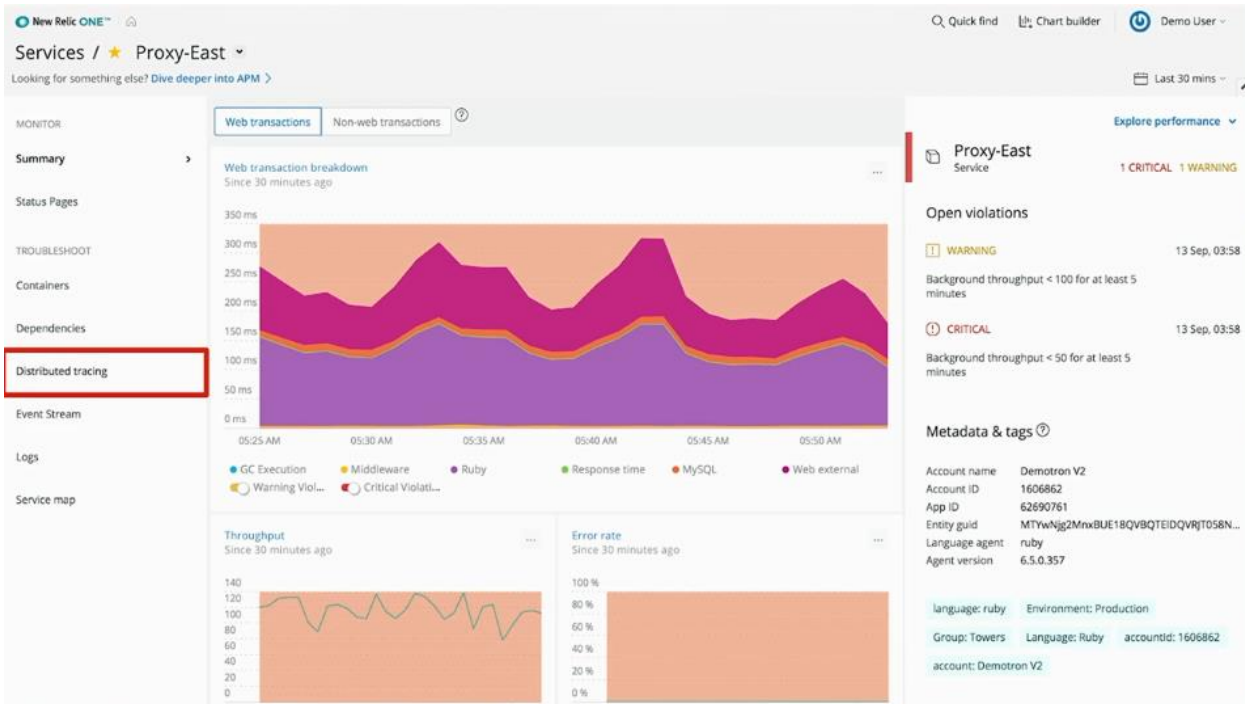
This could be a nerdlet within our platform



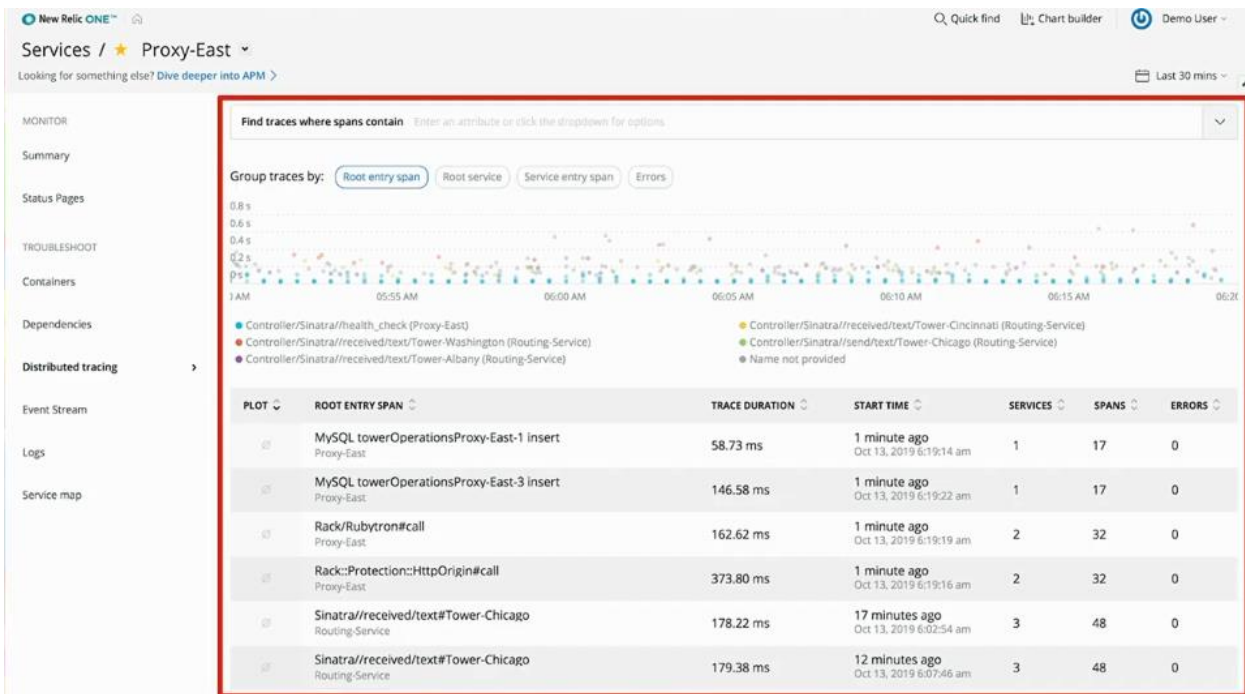
Above is how a micro-frontend is structure using 2 parts, the structure and the implementation parts.



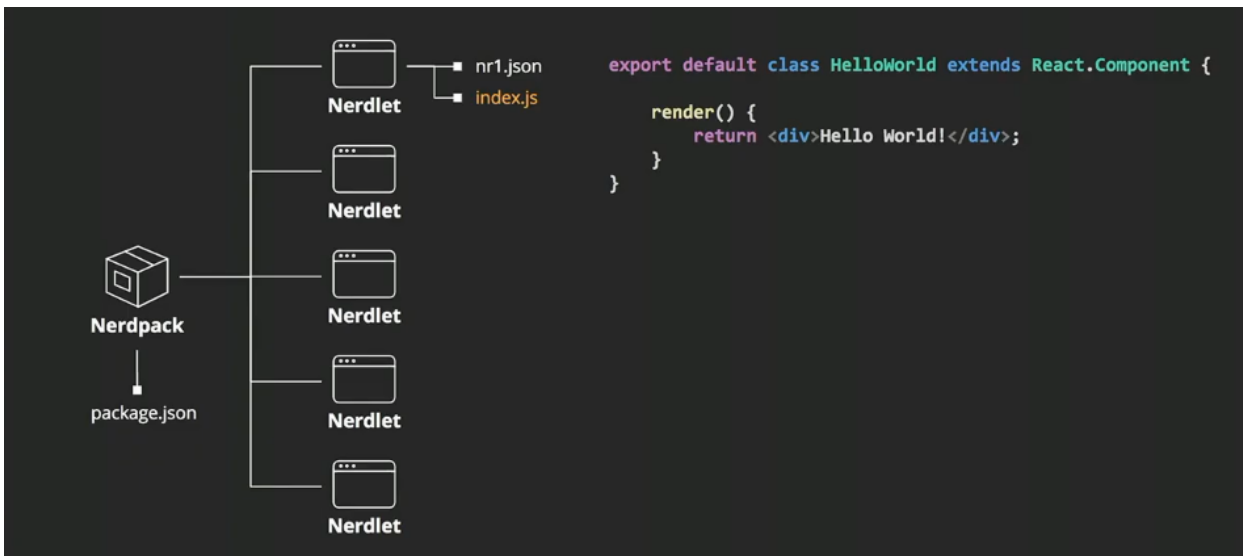
In the above platform UI, we have 2 nerdlets rendered. A side-bar nerdlet and a main view nerdlet.



This is a single nerdlet with a `displayName` of `distributedTracing`. It does not need to load its own JS or CSS, we just use its metadata information to build its UI.



When the UI clicks on the `distributedTracing` nerdlet, we then lazy-load the microfrontend and render it into the view

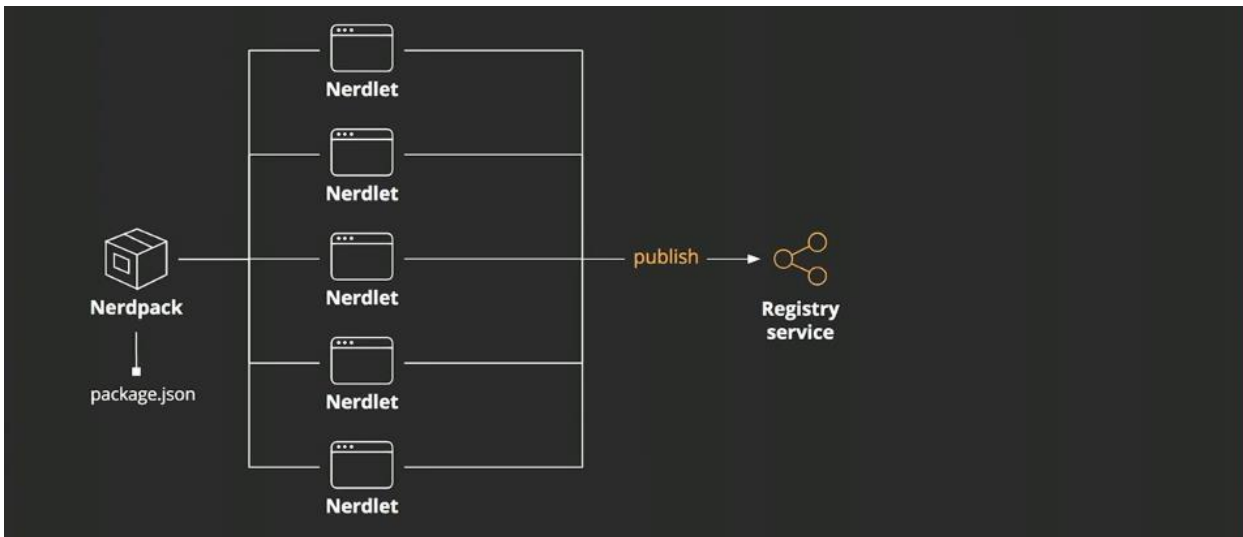


We use React for our rendering engine and above is how we render the micro-frontend's template and style files

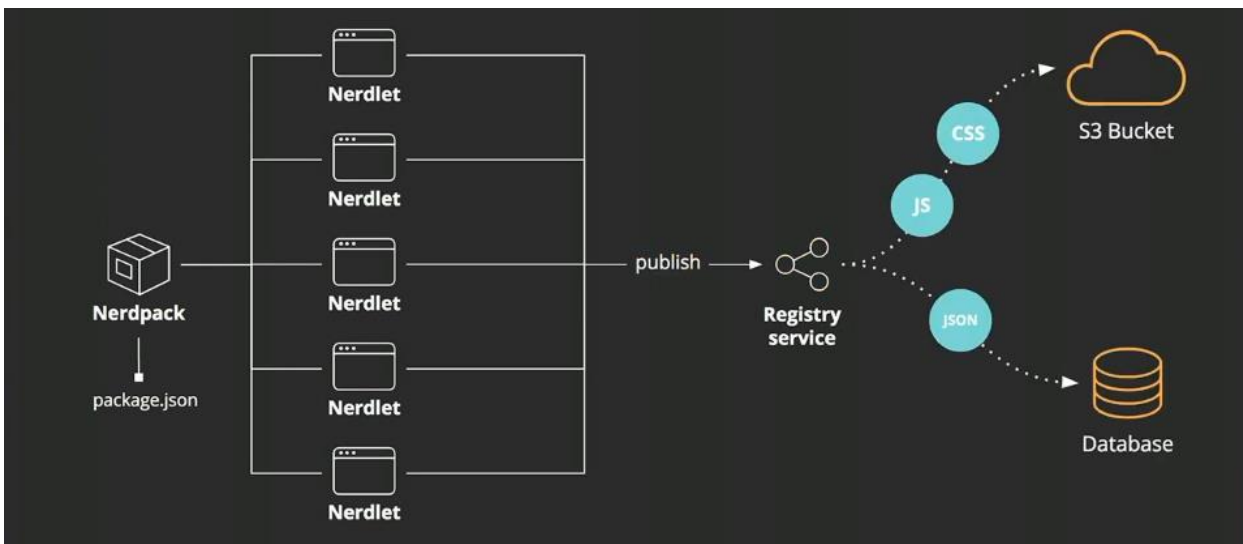


```
> nr1 create --type nerdpack --name my-nerdpack  
> cd my-nerdpack  
> nr1 create --type nerdlet --name hello-world  
> nr1 nerdpack:serve  
> nr1 nerdpack:publish
```

You can serve or publish a version of your nerdpack



When published, each micro-frontend in the nerdpack gets bundled separately as an app.



The CLI then calls a service registry that sends the static data (JS, CSS and JSON files) in S3 buckets and a database. We use this **Registry Service** to get artifacts for our cross-cutting UIs later via the metadata.

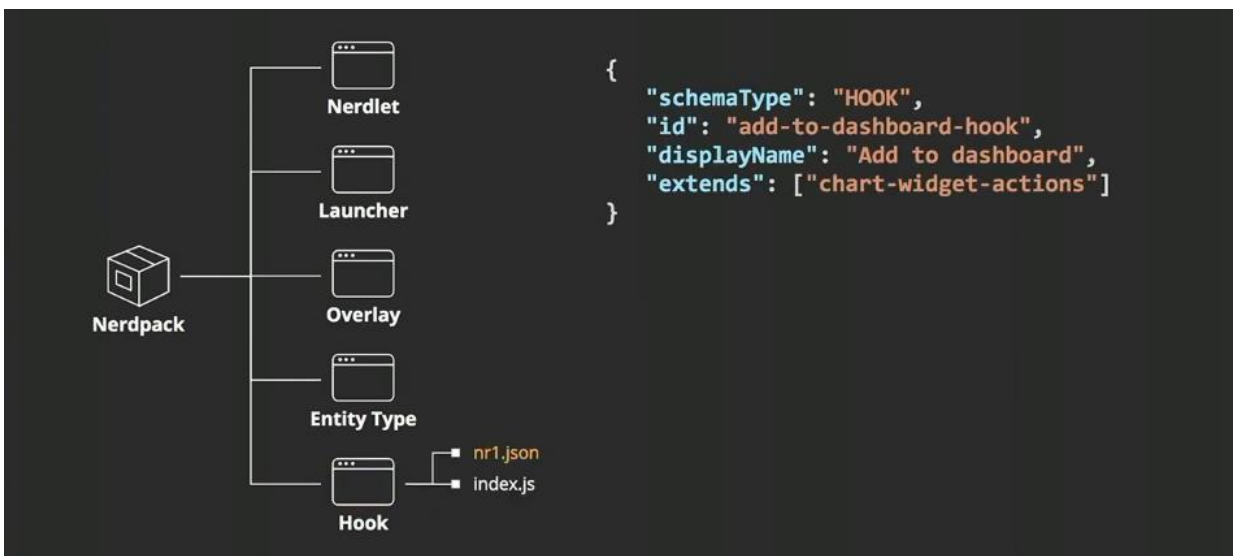
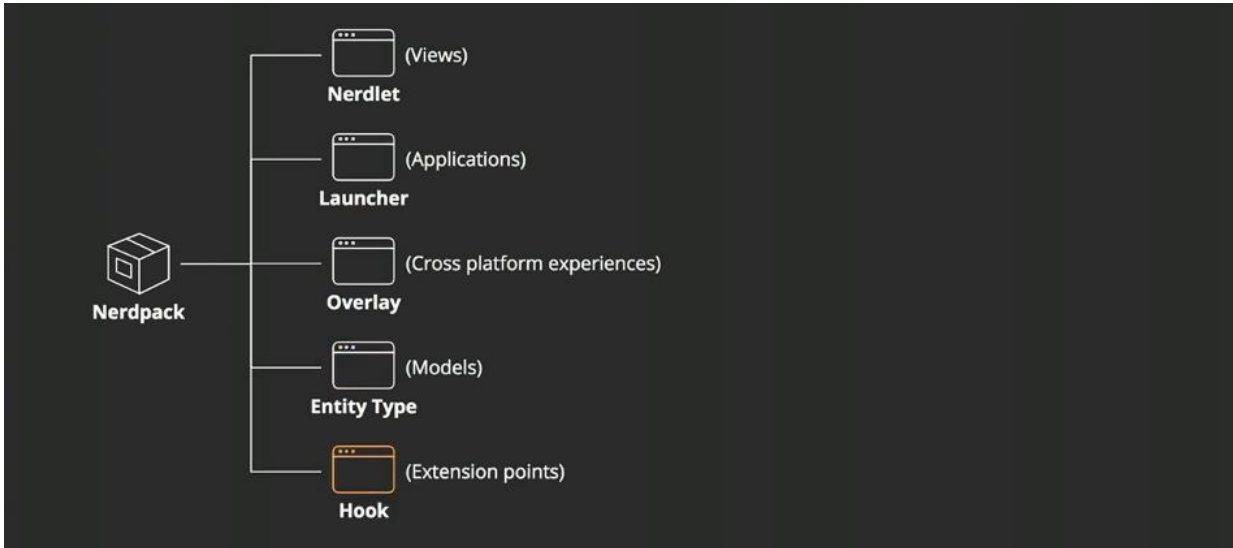
```
> nr1 create --type nerdpack --name my-nerdpack
> cd my-nerdpack
> nr1 create --type nerdlet --name hello-world
> nr1 nerdpack:serve
> nr1 nerdpack:publish
> nr1 nerdpack:deploy
```

We can also deploy the published version of the nerdpack to production

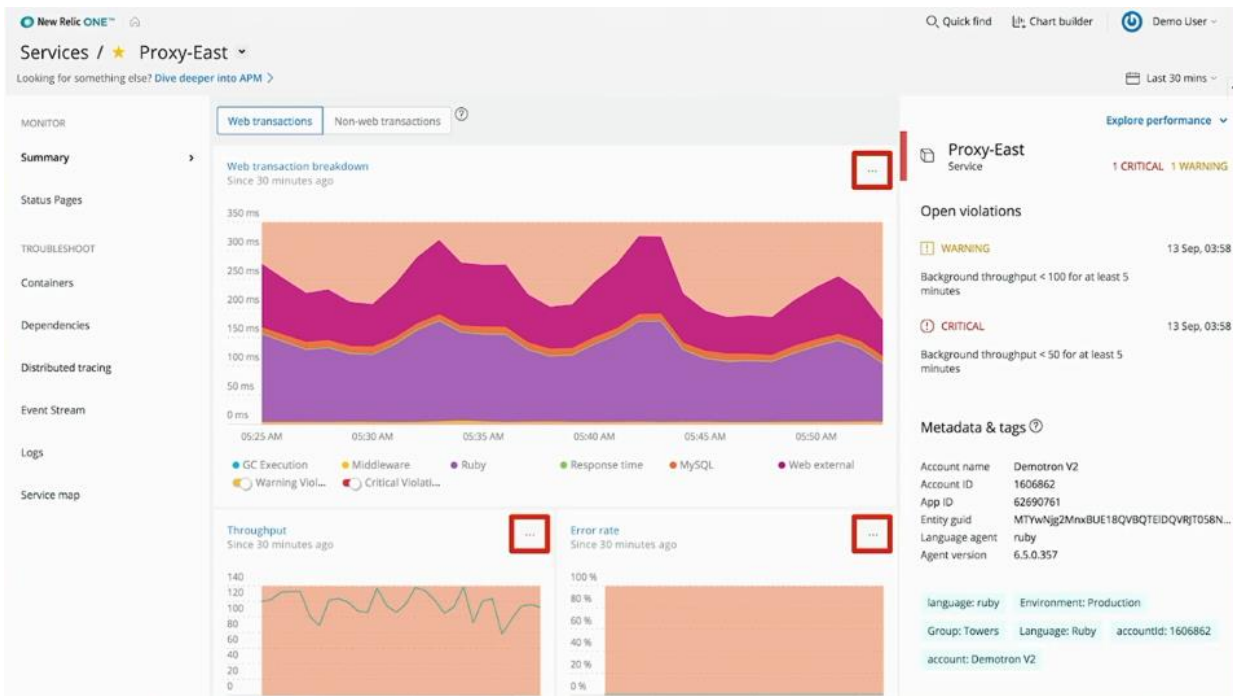
Everything works out of the box.

Other types of **micro frontends**.

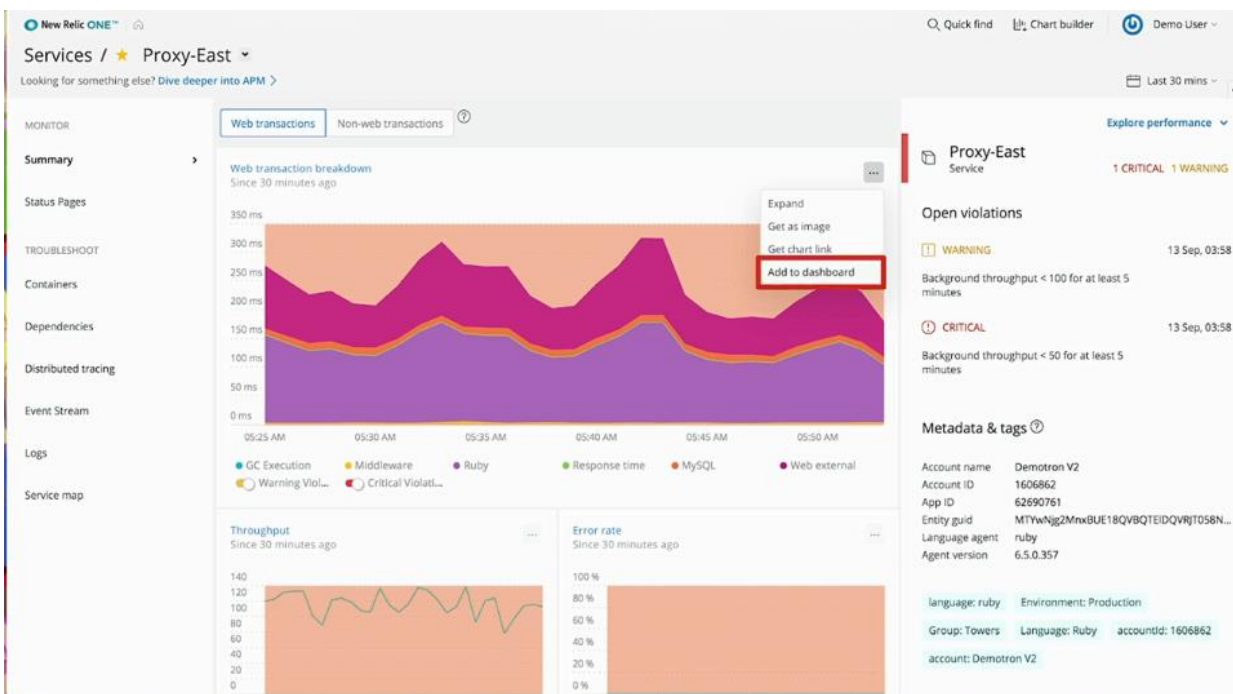
Most micro-frontends are UI components or fragments, but we have more types.



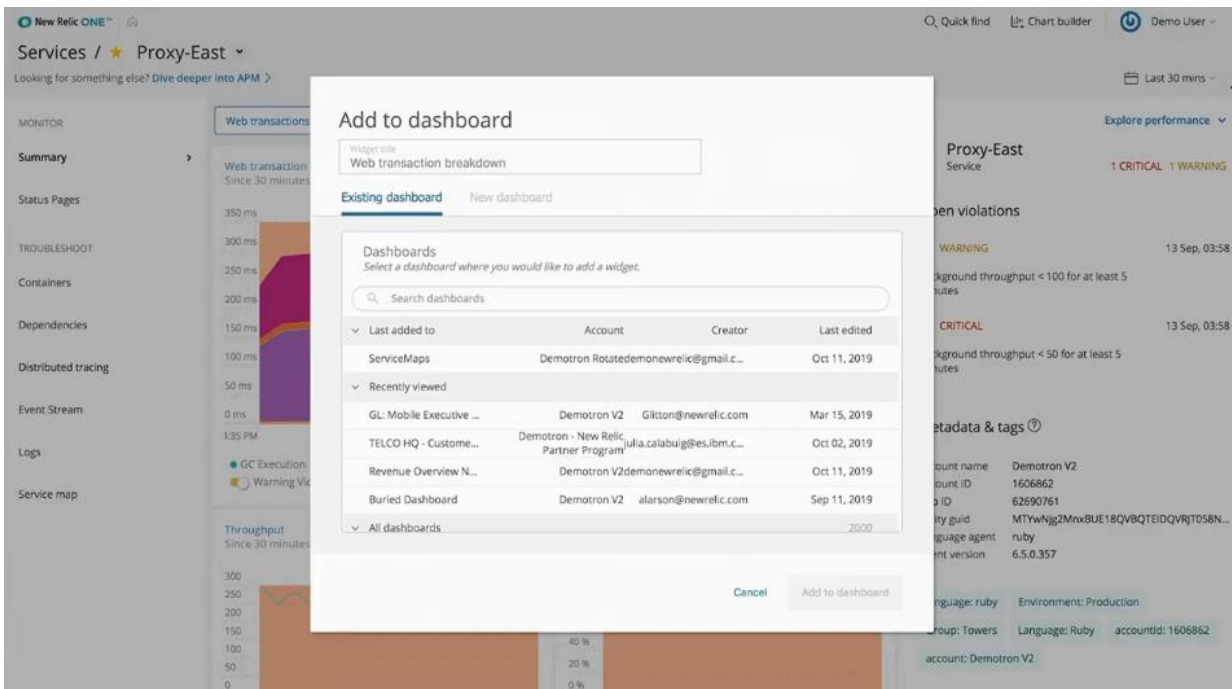
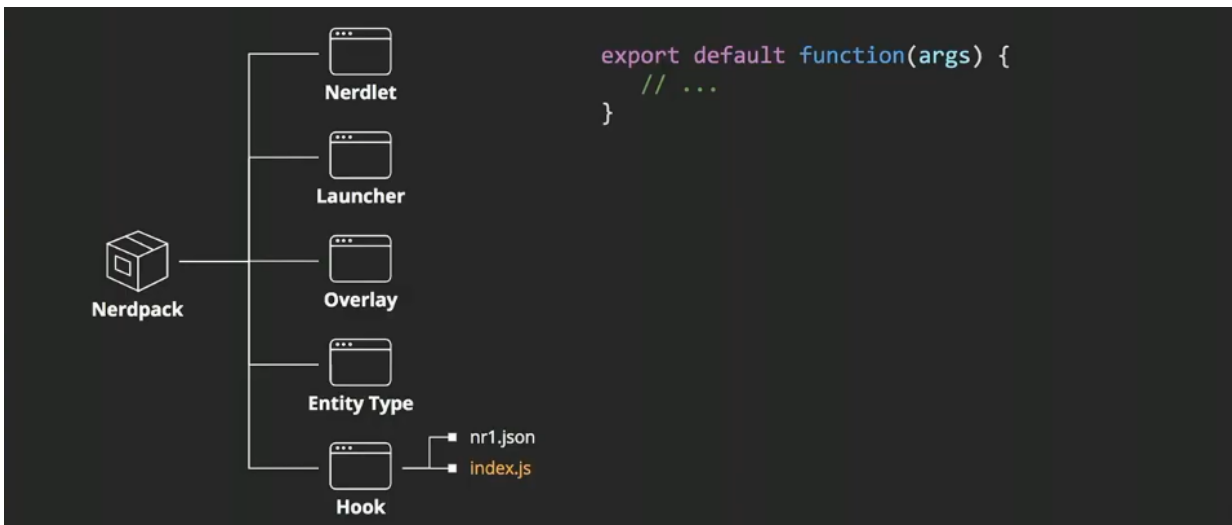
In this hook, we have the same metadata structure but we are also extending an extension point call chart-widget-actions.



New Relic is a monitoring tool with a lot of charts within the platform, each chart as a little clickable menu icon attached that the user can click to perform the for enlarging chart size action.

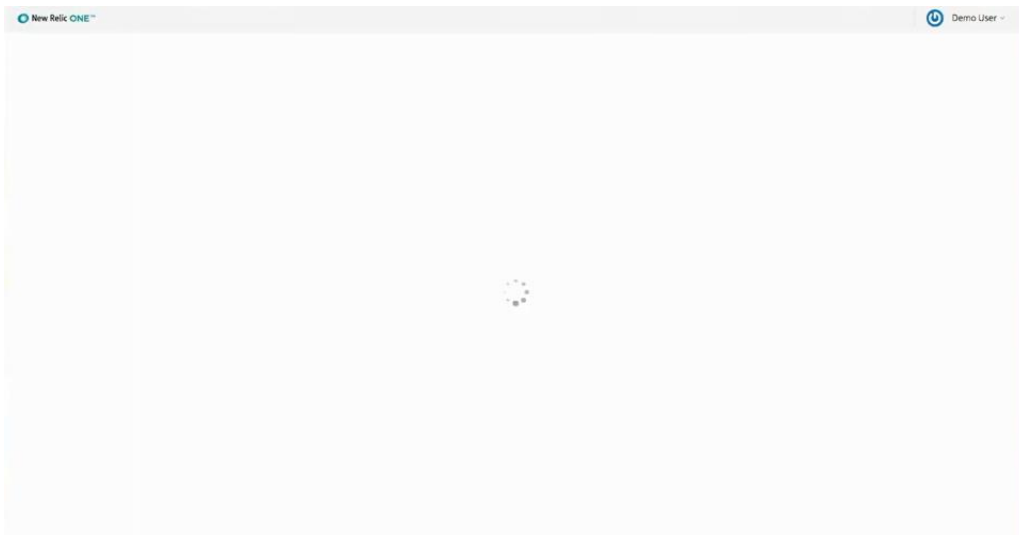


The chart then exposes the extension point that a hook can hook into, only when the user clicks on the hook do we load the implementation which is a function that receives some arguments as below.



The user can then add/enlarge the chart

Platform **application shell.**



This is basically the header, logo, user menu and a spinner.

Application shell responsibilities.

- Fast initial load.
- Handle cross cutting concerns.
- Layout composition.
- Provide common dependencies.

It handles cross-cutting concerns like the authentication, configuration data like getting user details/context, routing, service discovery

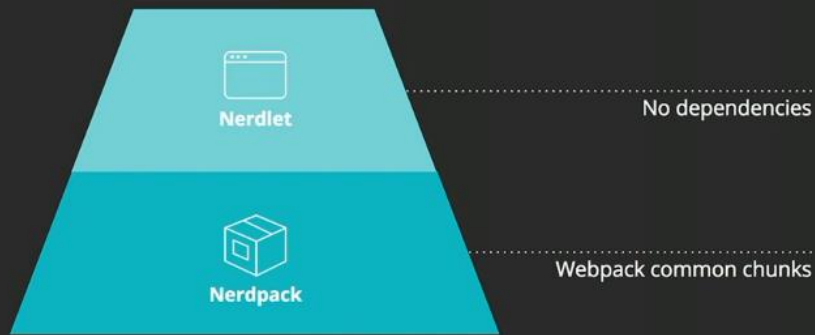
Dependency management.

Dependency deduplication.



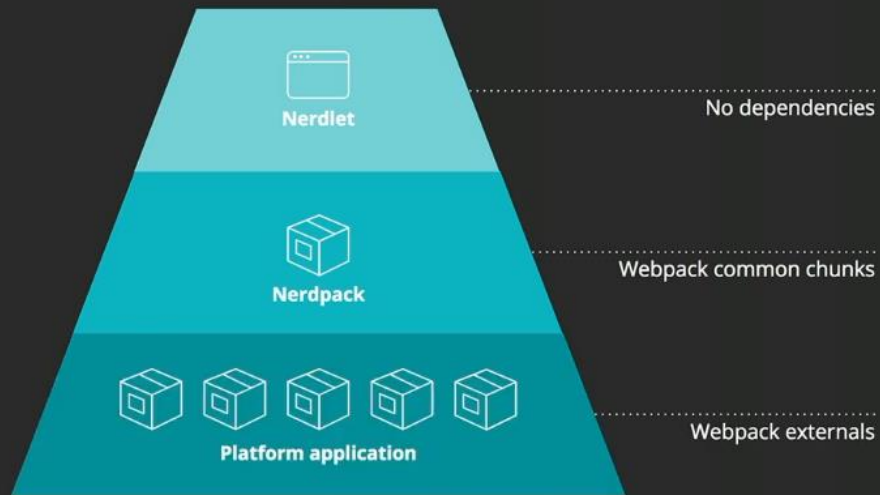
A micro-frontend in itself cannot define any dependencies, this detail is kept in its metadata on the higher level.

Dependency deduplication.



We instead define all the dependencies on the repository/nerdpack level, we dedupe all the node modules and share the common modules

Dependency deduplication.



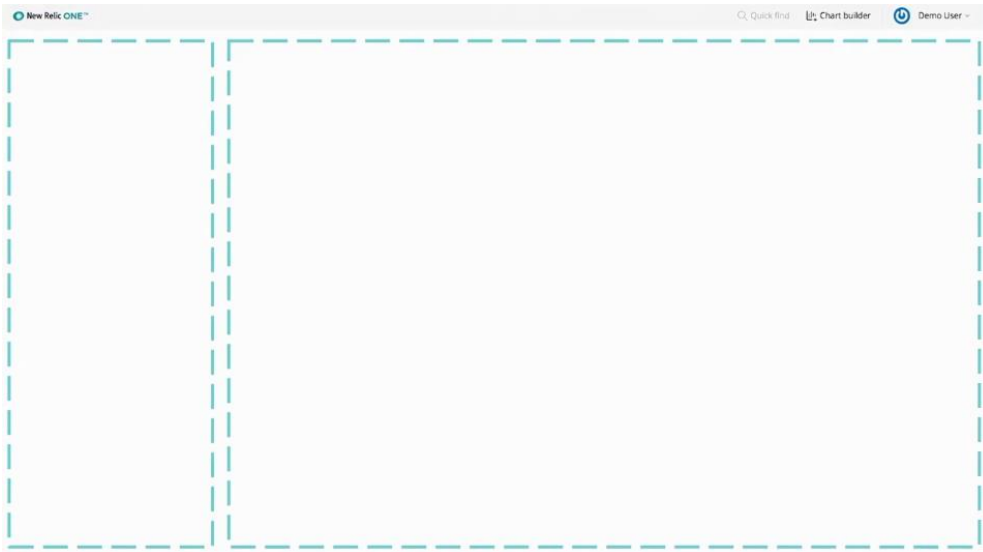
We then use Webpack external to dedupe between all the nerdpacks in the platform using the CLI

Sharing **common** libraries.

```
// webpack.config.js
module.exports = {
  externals: {
    'react': 'React',
    'react-dom': 'ReactDOM',
    'd3': 'd3',
  }
};
```

The externals then become the global nerdpack dependencies that the application downloads from a CDN.

Layout **composition**.



We define our layouts in different containers explicitly.

Layout **containers**.

- Loader mechanism.
- CSS namespacing.

We have a consistent loading mechanism, namespace our CSS by nerdpack name as below

CSS **namespacing**.

```
.my-nerdpack .HelloWorld {  
  background: #a1a1a1;  
}
```

Layout **containers**.

- Loader mechanism.
- CSS namespacing.
- Error boundaries.

We don't want rendering errors in one nerdlet to affect the platform

Component **error boundaries**.

```
class ErrorBoundary extends Component {  
  
  componentDidCatch(error, errorInfo) {  
    logError(error, errorInfo);  
  
    this.setState({  
      error,  
    });  
  }  
  
  render() {  
    if (this.state.error) {  
      return <ErrorMessage />;  
    }  
  
    return this.props.children;  
  }  
}
```

React has a lifecycle method called **componentDidCatch** that automatically gets triggered when an error occurs within a component, we use this to reset the state value to show some consistent error message view instead of an error.

Component **error boundaries**.

```
class NerdletContainer extends Component {  
  
  render() {  
    return (  
      <ErrorBoundary>  
        <Nerdlet />  
      </ErrorBoundary>  
    );  
  }  
}
```

We then wrap our nerdlet inside of the ErrorBoundary component to prevent the error leaking outside the container

Layout **containers**.

- Loader mechanism.
- CSS namespacing.
- Error boundaries.
- Basic logging / instrumentation.

Logging allows the teams to monitor their own micro-frontends



How do we decide what to render inside a container? We use routing to determine which specific nerdlet to render into the app shell container



We can also have more complex layouts like having a sidebar with a nerdlet inside it, we then can have 2 nerdlets displayed at the same time in the app shell view. We also have a state object that defines the specific layout that the user is seeing (users can also define the layout they want and save those layouts). All the URLs by default are permalinks that allow us to share the UI during troubleshooting.

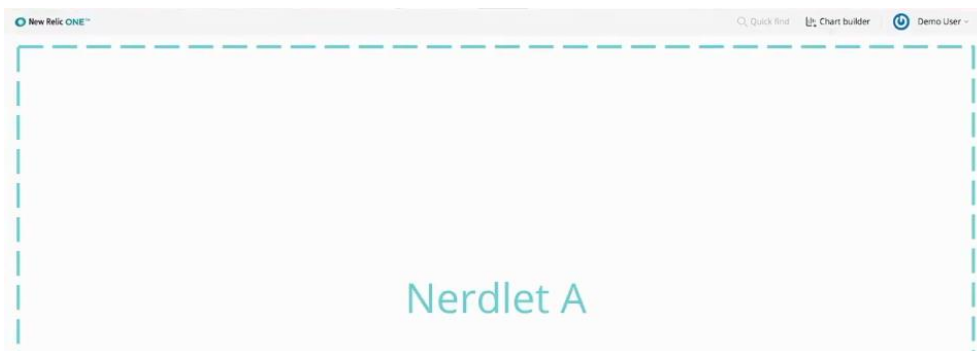
The platform **SDK**.

This is what we use to build experiences within the platform, it provides an API to configure the global platform features.

The platform **SDK**.

- Configure platform features.
- Navigation.
- Persist and receive URL state.
- Catalog of all micro frontends.
- Shared UI components.

Building **experiences**.



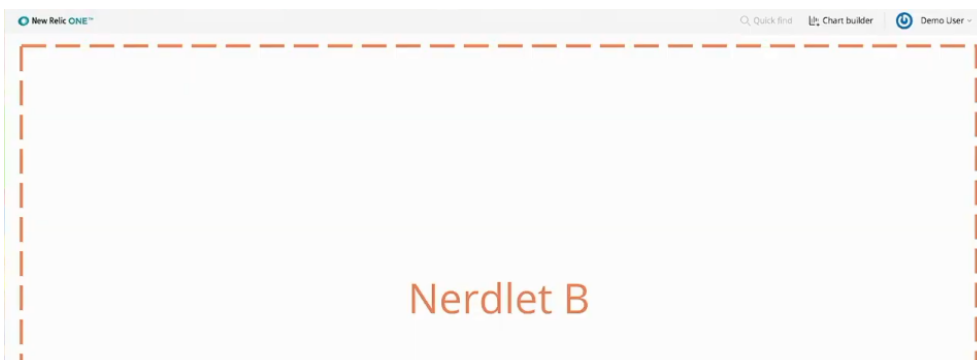
```
import { Button, navigation } from 'nr1';

export default class MyNerdlet extends React.Component {

  onClick() {
    navigation.openNerdlet({
      id: 'my-nerdpack.nerdlet-b',
      urlState: {
        entityId: 832974,
      }
    });
  }

  render() {
    return <Button onClick={this.onClick}>Click me</Button>;
  }
}
```

The nr1 is our SDK library



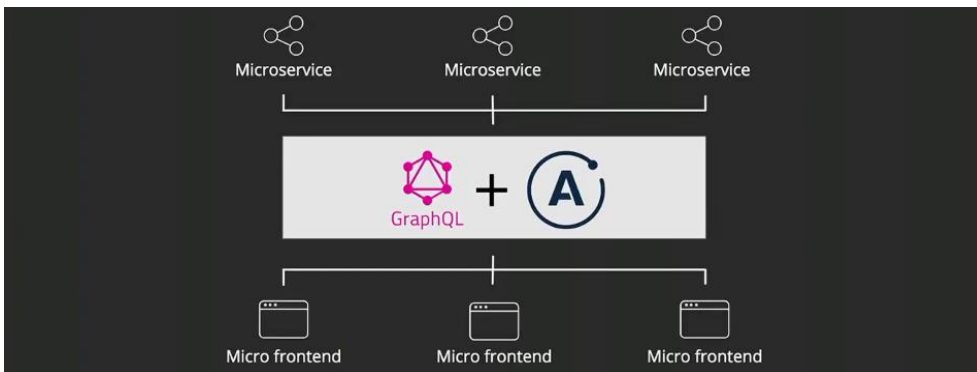
This then opens nerdlet B

Cross **communication.**

- URL State.
- Pub/Sub library.

We can use the URL state to communicate between the different micro-frontends.

Backend communication.



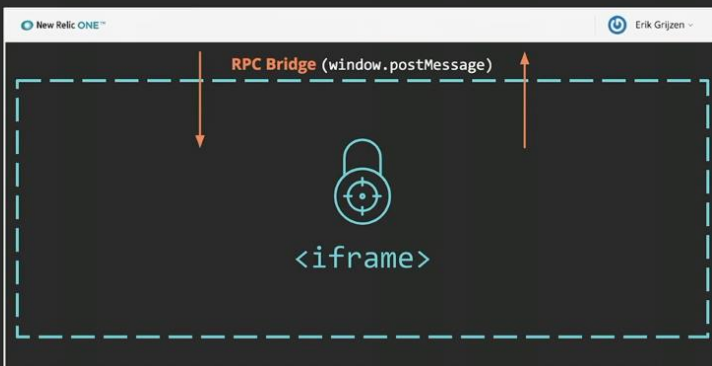
We use schema stitching so that teams can build their own smaller GraphQL schema that tend gets joined together for the main schema. The Apollo client helps to keep all data in sync after data fetches.

Third-party extensibility.

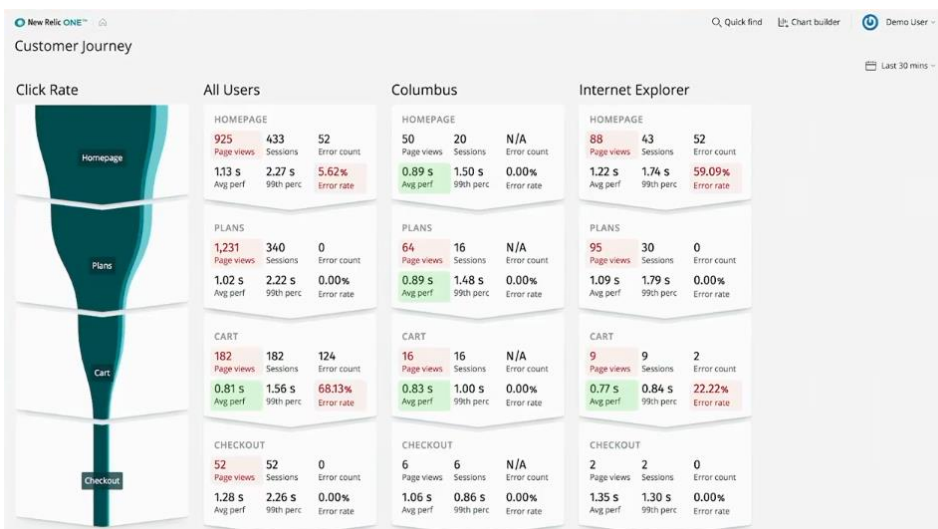
Third party **extensibility**.

- Open the CLI to the public.
- Secure sandboxing.
- Limited SDK surface.

Secure sandboxes.



We render the nerdlets inside an iframe to get the sandboxing effect, we then use an RPC bridge to pass events from/to the main application.



This is an extension built by a 3rd-party and it looks the same



Our experience

Did we achieve our **goals**?

- Easier and faster feature development.
- A more unified UI / UX.
- Improved performance.
- Third party extensibility.

How can we **scale**
our **UI development**?



1. **Communication** is still key.
2. Find the right **balance**.
3. Invest in **UI infrastructure**.

Thank You

 @ErikGrijzen

 bit.ly/2MrSE81