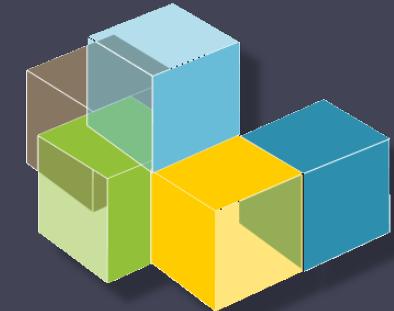


Universidad de Oviedo



Escuela de  
Ingeniería  
Informática



ARQUITECTURA  
DEL SOFTWARE

# Arquitectura del Software

Lab. 12

Monitorización y evaluación de rendimiento (profiling)

2020-21

José Emilio Labra Gayo  
Pablo González  
Irene Cid  
Paulino Álvarez

# Monitorización y profiling

**Monitorizar:** Observar comportamiento de un software

Cuadros de mando

Habitualmente, después del despliegue

**Profiling (caracterizar):** Medir rendimiento de un software mientras se ejecuta

Identificar partes que contribuyen a un problema

Mostrar dónde centrar los esfuerzos para mejorar rendimiento

Suele hacerse antes del despliegue

Monitorizar una aplicación mientras se ejecuta  
Registrar uso de CPU, memoria, hilos, etc.

JavaScript:

Chrome (Timeline), Firefox Developer Edition  
(Performance tool)

Herramientas de servidor:

JVisualVM, JProfiler, YourKit, Jconsole, etc.

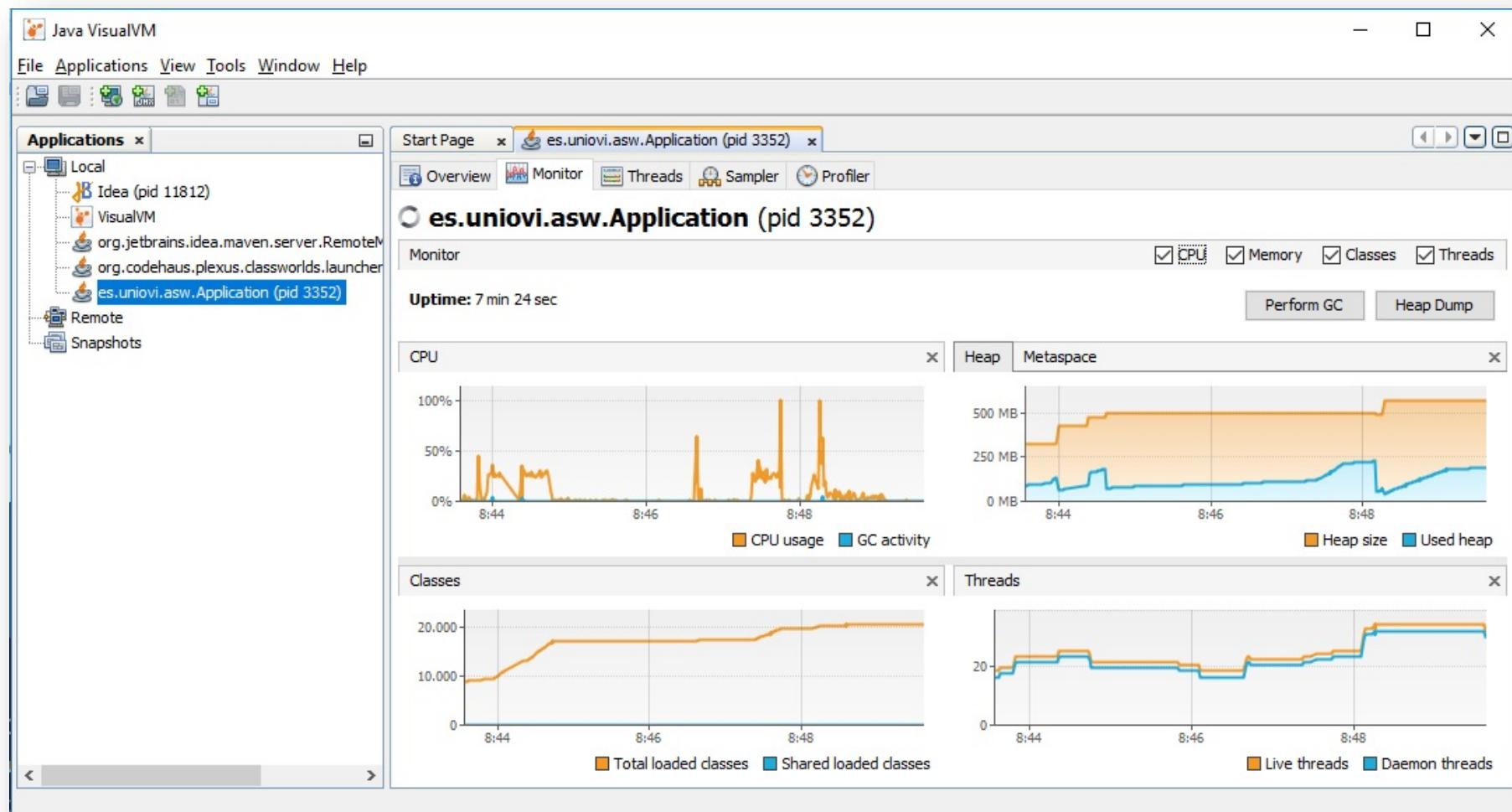
Graphite, Datdog, Prometheus, Graphana

VisualVM

<https://visualvm.github.io/>

Ya está instalada con el JDK: jvisualvm

# Server/Java: JVisualVM



# Navegador: developer tools

- Monitorizar/chequear rendimiento

The screenshot shows the 'Tools for Web Developers' website with the 'Chrome DevTools' tab selected. On the left, a sidebar lists various developer tools: Home, Open DevTools, CSS, Console, Network, Storage, Command Menu, Mobile Simulation, DOM, JavaScript, and Performance. Under 'Performance', there is a 'Get Started' section. The main content area displays a 'Get Started With Analyzing Runtime Performance' article by Kayce Basques, a Technical Writer for Chrome DevTools & Lighthouse. The article includes a small profile picture of Kayce and a performance timeline visualization showing various tasks and their execution times. To the right of the article is a 'Contents' sidebar with links to 'Get started', 'Simulate a mobile CPU', 'Set up the demo', 'Record runtime performance', 'Analyze the results', 'Analyze frames per second', 'Find the bottleneck', 'Bonus: Analyze the optimized version', and 'Next steps'. At the bottom of the page, there are navigation links for 'Console', 'What's New', and a close button.

<https://developers.google.com/web/tools/chrome-devtools/evaluate-performance>

# Navegador Ejemplo: Google Chrome

## Modo incognito

En la esquina superior derecha, click en los tres puntos y nueva ventana incógnito

Windows, Linux, or Chrome OS: Ctrl + Shift + n.

Mac: ⌘ + Shift + n.

## Chrome DevTools

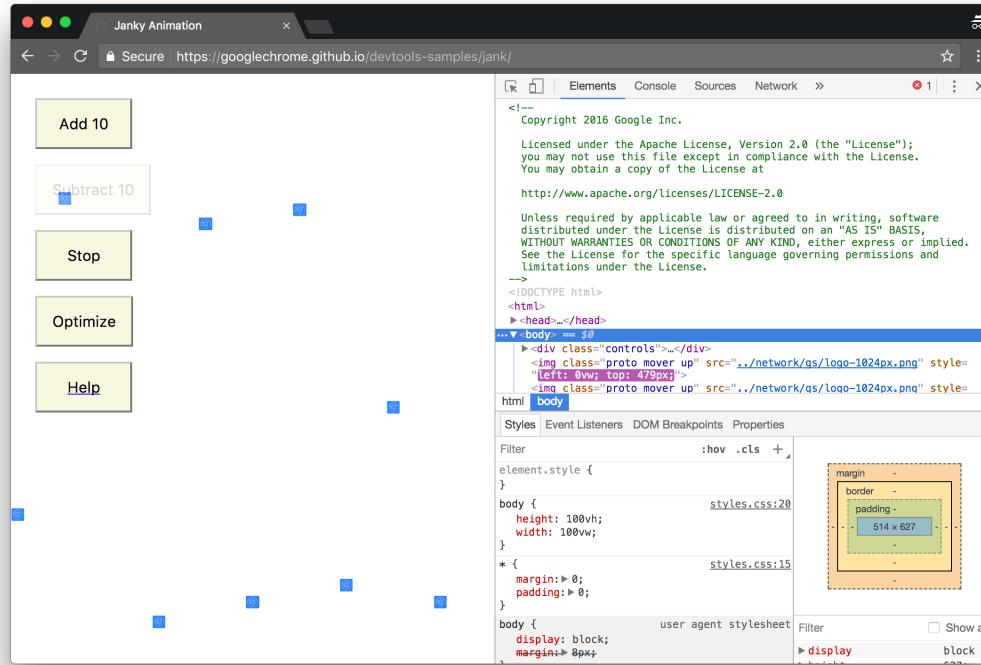
Windows, Linux: Control+Shift+I

Mac: Command+Option+I



# Navegador Ejemplo: Google Chrome

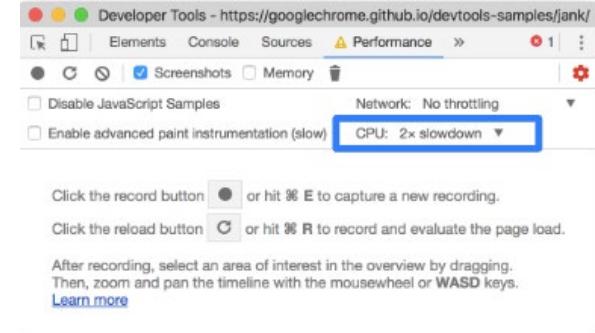
<https://googlechrome.github.io/devtools-samples/jank/>



Performance>Record  
click Add 10 (20 veces)  
Optimize / Un-optimize  
Stop



Performance>CPU>2 x Slowdown

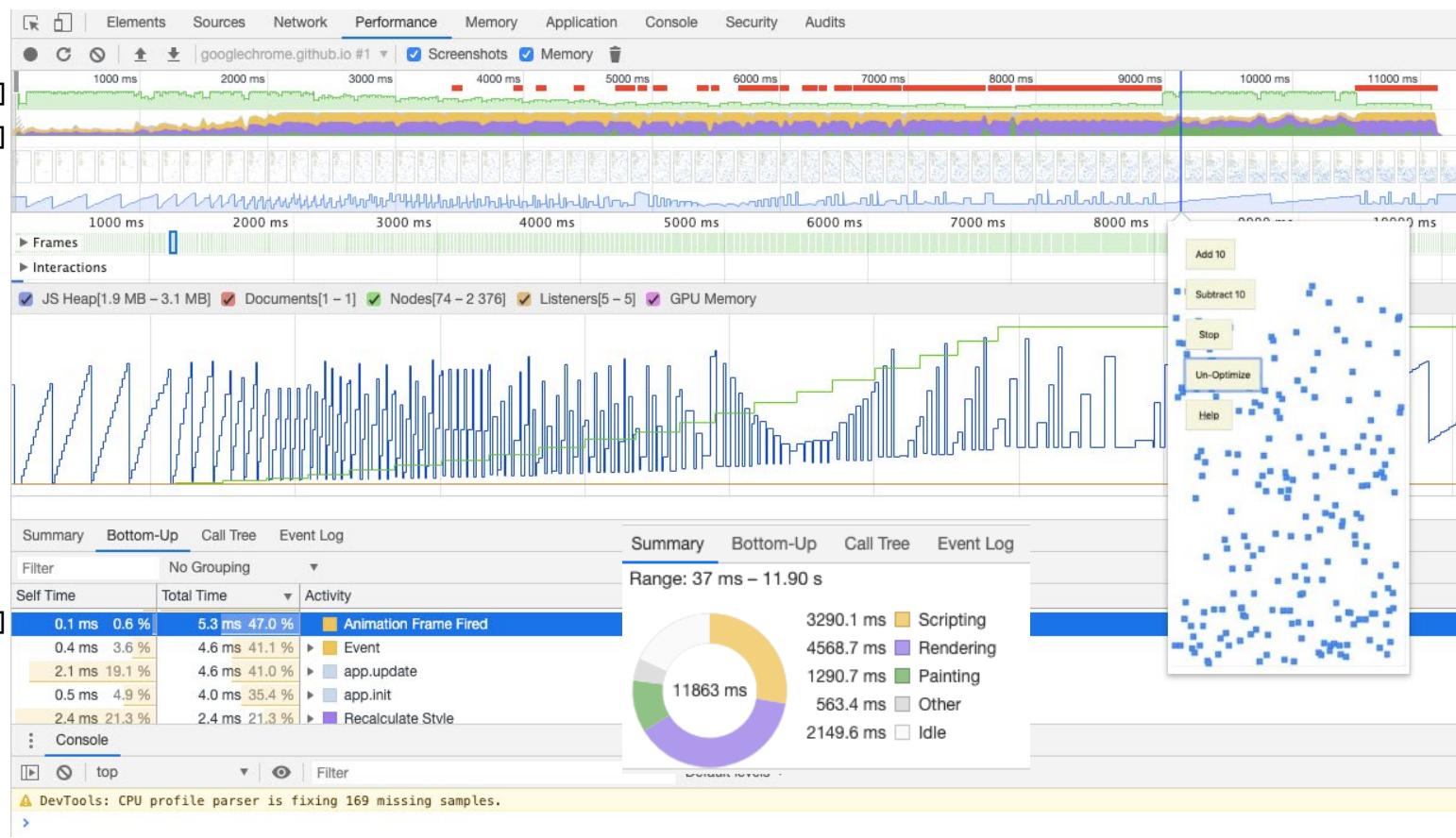


# Navegador Ejemplo: Google Chrome

Profile:

Frames por Segundo → □

CPU → □



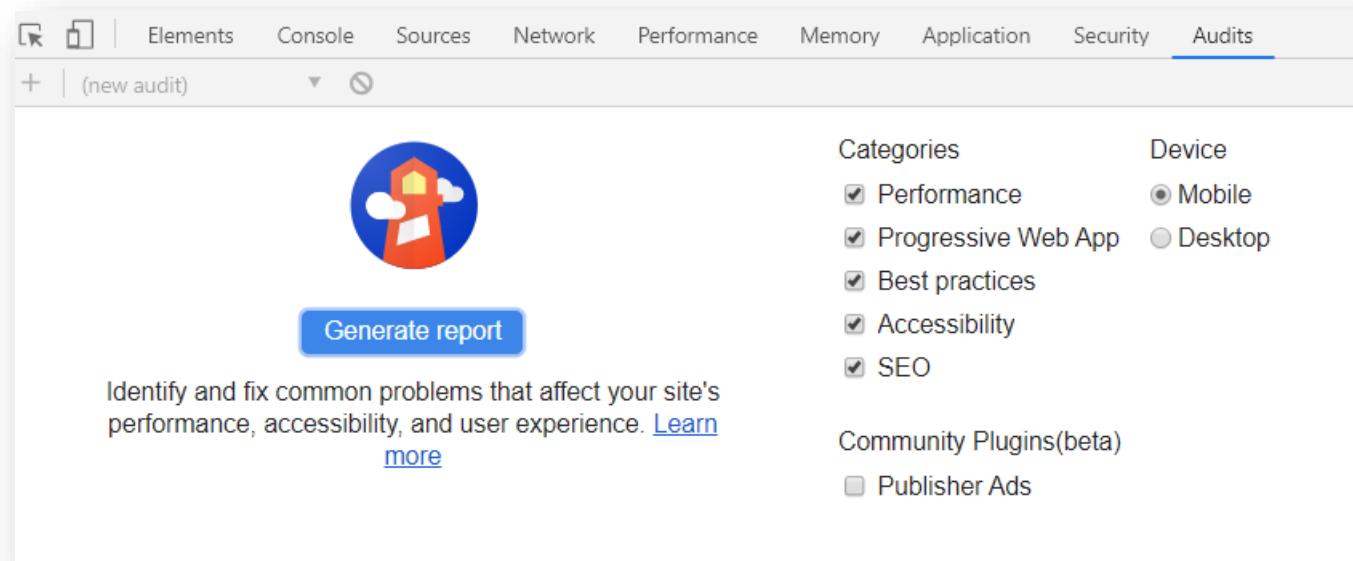
# Otras herramientas de navegador

## RAIL model (Response, Animation, Idle, Load)

<https://developers.google.com/web/fundamentals/performance/rail>

<https://webpagetest.org/easy>

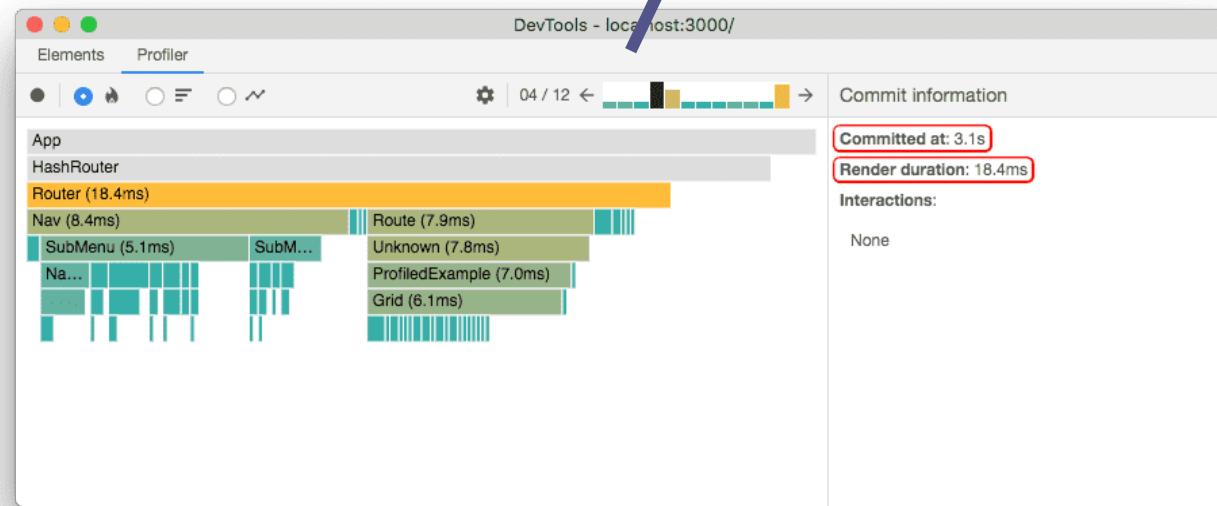
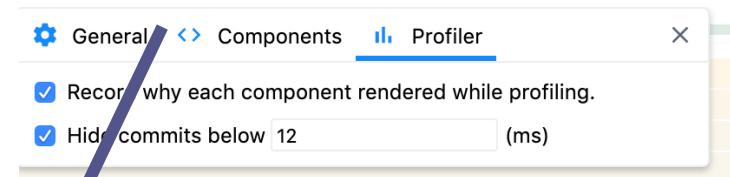
Lighthouse (viene con Chrome)



# React Herramientas desarrollo

React trabaja en dos fases:

- Render
- Commit

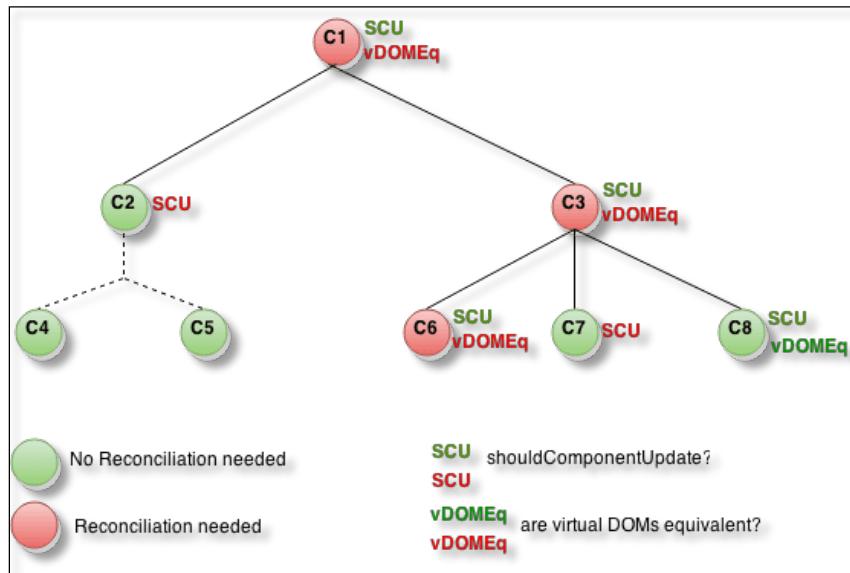


# React Herramientas desarrollo

The screenshot shows the inrupt Profiler tool interface. On the left, there are input fields for address ('avda galicia'), locality ('Oviedo'), postal code ('33005'), and region ('Asturias'). The toolbar at the top includes tabs for General, Components, Profiler (which is selected), and Audits. Below the toolbar, there are two checked checkboxes: 'Record why each component rendered while profiling.' and 'Hide commits below 12 (ms)'. The main area features a Flamegraph visualization of component rendering times. A purple oval highlights the entry for 'Xd key="subject:\_:UserProfileAddress\_\_parts\_4"' with a duration of 0.2ms. This entry is expanded to show its render details, which include:

- Why did this render?**
  - Props changed: (fieldData, modifyFormObject, formObject, onSave)
- Rendered at:**
  - 4.1s for 17.3ms
  - 4.5s for 83.8ms
  - 4.6s for 19.9ms
  - 4.7s for 19.9ms
  - 5s for 16.1ms** (highlighted in blue)
  - 6.6s for 21.4ms

# React: DOM - Virtual DOM



```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

```
shouldComponentUpdate(nextProps, nextState) {
  if (this.props.color !== nextProps.color) {
    return true;
  }
  if (this.state.count !== nextState.count) {
    return true;
  }
  return false;
}
```

# Monitorización en servidor

- Las plataformas en la nube brindan soluciones de monitoreo
  - También disponible en Google Cloud, Amazon AWS, ...
  - En el caso de Heroku, esta solución no es gratuita
- Aunque también existen soluciones de terceros
  - Prometheus, Graphite, Grafana, Datadog, Nagios, Sensu, ...
- Usaremos: **Prometheus** y **Grafana**
  - Radarin: [https://github.com/arquisoft/radarin\\_0/tree/master/restapi#monitoring-prometheus-and-grafana](https://github.com/arquisoft/radarin_0/tree/master/restapi#monitoring-prometheus-and-grafana)



## Grafana   Prometheus

- **Prometheus:** servidor de almacenamiento de datos en series de tiempo
  - Modelo de datos multidimensional
  - Lenguaje flexible de consultas
  - Nodos autónomos de servidor único
  - Configuración estática
- **Grafana:** Visualización de datos. Permite crear, explorar y compartir tableros

# Monitorización en servidor

- Necesitamos una biblioteca que pueda extraer algunas métricas de nuestro restapi

1. Instalar el cliente

```
npm install prom-client express-prom-bundle
```

2. Modificamos *restapi/server.js*

```
//Monitoring middleware
const metricsMiddleware = promBundle({includeMethod: true});
app.use(metricsMiddleware);
```

3. Si lanzamos el restapi, en */metrics* podremos ver algunos datos de fila que Graphana usaría para trazar buenos gráficos.

Podemos elegir que métrica medir [\[doc\]](#)



Grafana Prometheus

# Monitorización en servidor

- Graphana no puede usar esta información directamente, necesita Prometheus
  - Prometheus recuperará los datos expuestos por el restapi y los almacenará para que Grafana pueda consumirlos.
  - Trabajaremos con una docker image [prom/prometheus] que se puede configurar a través de un solo archivo

```
restapi > monitoring > prometheus > ! prometheus.yml
1 global:
2   scrape_interval: 5s
3   scrape_configs:
4     - job_name: "example-nodejs-app"
5       static_configs:
6         - targets: ["restapi:5000"]
```



Grafana



Prometheus

# Monitorización en servidor

- Como configurar Graphana
  - Graphana usará Prometheus como fuente de datos
  - Tenemos una docker image para ejecutarlo [grafana/grafana]
  - Nosotros necesitamos configurar datasource y el dashboard (gráficos a visualizar)



Grafana



Prometheus

# Referencias

- Monitorización y Profiling

- Get Started With Analyzing Runtime Performance

<https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/>

- How to Use the Timeline Tool

<https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/timeline-tool#profile-js>

- Otro Ejemplo

<https://github.com/coder-society/nodejs-application-monitoring-with-prometheus-and-grafana>