To Kill a Monolith: Slaying the Demons of a Monolith with Node.js Microservices on Cloud Foundry

Tony Erwin, Senior Software Engineer, IBM
aerwin@us.ibm.com
Agenda

• Origins of the Bluemix UI
• Demons of the Monolith
• Slaying Demons with a Microservices Architecture
• New Demons to Slay
Bluemix UI – Origins as a Monolith

- Serves as front-end to Bluemix (IBM’s open cloud offering)
- Lets users create, view, and manage CF resources, containers, virtual servers, user accounts, billing/usage, etc.
- Runs on top of Bluemix PaaS Layer (Cloud Foundry)
- Started as single-page application (SPA) to provide desktop-like experience in the browser
  - All HTML, CSS, and JavaScript loaded within single web page
  - Served from a single Java app
- State-of-the-art not all that long ago
  - Dojo + J2EE was the most common stack in IBM when Bluemix UI dev started (~4 years ago)
  - SPA still popular (AngularJS, Ember.js etc.)
Bluemix UI – Many Components

Home
Dashboard
Resource Details
Users
Billing
And More!
Monolithic Architecture

Cloud Foundry

Bluemix UI Server (Java)

DB/2

Backend APIs (CF, Containers, VMs, Billing, Authentication, etc.)
Demons of the Monolith

- Bad performance
  - Heavy weight JavaScript loaded to browser is slow
  - Volume of client-initiated AJAX requests creates bottlenecks
- Difficult to integrate code from other teams (especially when they use different stacks)
- Have to push whole product even for small changes
- Poor SEO
- New hires want nothing to do with Dojo
Bluemix UI Microservices Architecture
Weapons of Microservices

• Aids migration to more modern, lighter-weight stack without starting over
• Improves performance with small services optimized for speed and page size
• Increases developer productivity with less chance of breaking other parts of the product
• Loosely-coupled services can deploy at their own schedule
  • Teams use stack of their choosing
  • Teams don’t have to wait on others
• Leads to improved SEO
  • Proxy facilitates “clean” URLs
  • Server side generation results in crawlable content
• Improves cross-team UI consistency via microservice composition
Microservice Pattern

UI microservice:
- Written with Node.js
- Serves lightweight HTML, CSS, JS (simplest approach that works)
- Uses server-side templating (Dust.js) to make as much data available to the client as possible for fast rendering & SEO
- Invokes Common Header microservice to get HTML for shared header to include in initial payload
- Consults shared session store for user token, etc.
- Leverages backend APIs and/or API microservices as appropriate
Page Composition w/ Microservices

- header
- inline
- combined

Diagram showing a microservices architecture with common microservices connected to specific instances and resources.
Migration, Phase 1

- Delivered Feb 2015
- Introduced proxy layer to route requests to different services based on URL path
- Added three microservices behind the proxy (alongside Java code)
- Leveraged two additional CF services:
  - Data Cache for shared session
  - NoSQL DB for data storage
Migration, Phase 2

- Delivered Feb 2016
- 90%+ of migration complete
Migration End Goal

- More or less current state in production, except for:
  - Java app for certain APIs still exists
  - Small amount of legacy Dojo code
Plugin Architecture

- Roughly 15 teams outside of the core UI team plug-in via the proxy to look and behave like one large product.
New Demons to Slay
New Demons to Slay

• More moving parts, more complexity
  • Build pipeline becomes all the more important
• Collecting federated status, monitoring health of the system
• Granularity of microservices vs. memory allocation
  • Monolith: 3 instances of Java app = 6 GB
  • New console: ~27 apps, ~95 instances, ~55.5 GB
• Seamless navigation with existing monolith
• Blue-green deployments
• Promoting uniformity & consistency while still giving teams freedom
• Geo-load balancing and failover
  • Taking lessons learned with resiliency in one Cloud Foundry deployment and applying globally across several
Importance of Monitoring

• We quickly learned that you can’t run a microservice-based system without monitoring
  • Lots of things can go wrong
  • Root cause determination can be difficult

• Examples of needed metrics:
  • Data for every inbound/outbound request for every microservice
    • Response time
    • Response code
  • Memory usage, CPU usage, and uptime for every microservice
  • General health of ourselves and dependencies
  • Synthetic page load data with Sitespeed.io & WebPageTest
Global Console: Geo Load Balancing and Failover

• Global Console Phase 1 went live in Summer 2017

• One global URL (console.bluemix.net)

• UI region switcher filters the view rather than redirect to different URL

• Use Dyn geo load balancing to serve UI from the nearest healthy region

• If healthcheck in a region shows a problem, Dyn routes to the next closest healthy region

• Odds of all four regions being down at the same time much less than one region being down
Questions?

Tony Erwin
Email: aerwin@us.ibm.com
Twitter: @tonyerwin