

Hands-on with TicketMonster

Openshift UI Navigation

Red Hat
OpenShift Container Platform

Project: project-98 ▾ Application: all applications ▾

Developer

Administrator

Developer

Add

No workloads found

To add content to your project, create an application, component or service using one of these options.

From Git

Import code from your git repository to be built and deployed

Container Image

Deploy an existing image from an image registry or image stream tag

From Catalog

Browse the catalog to discover, deploy and connect to services

From Dockerfile

Import your Dockerfile from your git repo to be built & deployed

YAML

Create resources from their YAML or JSON definitions

Database

Browse the catalog to discover database services to add to your application

Openshift UI Navigation

The screenshot shows the OpenShift Container Platform UI with the following interface elements:

- Header:** Red Hat OpenShift Container Platform logo, navigation menu icon, project dropdown (Project: project-98), application dropdown (Application: all applications), a plus sign icon, a question mark icon, and user98 dropdown.
- Left Sidebar:** Developer mode selected. Options include +Add (highlighted with a blue underline), Topology, Builds, and Advanced.
- Central Content:** Title "Add" and message "No workloads found". Sub-instruction: "To add content to your project, create an application, component or service using one of these options."
- Deployment Options:** A grid of five cards:
 - From Git:** Import code from your git repository to be built and deployed.
 - Container Image:** Deploy an existing image from an image registry or image stream tag.
 - From Catalog:** Browse the catalog to discover, deploy and connect to services.
 - From Dockerfile:** Import your Dockerfile from your git repo to be built & deployed.
 - YAML:** Create resources from their YAML or JSON definitions. This card is highlighted with a red border.
- Bottom Card:** **Database**: Browse the catalog to discover database services to add to your application.

Openshift UI Navigation

The screenshot shows the OpenShift Container Platform UI with the following interface elements:

- Header:** Red Hat OpenShift Container Platform logo, navigation menu, project dropdown (Project: project-98), application dropdown (Application: all applications), and user information (user98).
- Left Sidebar:** Developer mode selected. Other options include +Add (highlighted with a blue underline), Topology, Builds, and Advanced.
- Central Content:** The "Add" screen. It displays a message: "No workloads found" and "To add content to your project, create an application, component or service using one of these options."
- Deployment Options:** A grid of five cards:
 - From Git:** Import code from your git repository to be built and deployed.
 - Container Image:** Deploy an existing image from an image registry or image stream tag.
 - From Catalog:** Browse the catalog to discover, deploy and connect to services. This card is highlighted with a red border.
 - From Dockerfile:** Import your Dockerfile from your git repo to be built & deployed.
 - YAML:** Create resources from their YAML or JSON definitions.
- Bottom Card:** Database, which allows browsing the catalog to discover database services to add to your application.

Openshift UI Navigation

Red Hat OpenShift Container Platform user98 ▾

Developer

+Add

Topology

Builds

Advanced

Project Details

Project Access

Metrics

Search

Events

Project: project-98

Developer Catalog

Add shared apps, services, or source-to-image builders to your project from the Developer Catalog. Cluster admins can install additional apps which will show up here automatically.

All Items

Filter by keyword...

108 items

Languages

Databases

Middleware

CI/CD

Other

TYPE

Service Class (0)

Template (97)

Source-to-Image (11)

Installed Operators (0)

.NET

.NET Core

Build and run .NET Core 3.0 applications on RHEL 7. For more information about using this builder image, including OpenShift considerations.

.NET Core + PostgreSQL (Persistent)

An example .NET Core application with a PostgreSQL database. For more information about us.

.NET Core Example

An example .NET Core application.

3scale APICast API Gateway

provided by Red Hat, Inc.

3scale's APICast is an NGINX based API gateway used to integrate your internal and external API services with

Apache HTTP Server

provided by Red Hat, Inc.

An example Apache HTTP Server (`httpd`) application that serves static content. For more information about us.

Apache HTTP Server (`httpd`)

provided by Red Hat, Inc.

Build and serve static content via Apache HTTP Server (`httpd`) 2.4 on RHEL 7. For

CakePHP

CakePHP + MySQL (Ephemeral)

provided by Red Hat, Inc.

An example CakePHP application with a MySQL database. For more information about using th

php

CakePHP + MySQL

(Ephemeral)

provided by Red Hat, Inc.

An example CakePHP application with a MySQL database. For more

Openshift UI Navigation

The screenshot shows the OpenShift Container Platform UI with the following interface elements:

- Header:** Red Hat OpenShift Container Platform logo, navigation menu icon, user icon (user98), and help/question icons.
- Project Selector:** Project: project-98 dropdown.
- Sidebar:** Developer tab selected, followed by +Add, Topology, Builds, Advanced (with a dropdown arrow), Project Details, Project Access, Metrics, Search, and Events.
- Main Content Area:** **Developer Catalog** heading. A note says: "Add shared apps, services, or source-to-image builders to your project from the Developer Catalog. Cluster admins can install additional apps which will show up here automatically." A "2 items" badge is visible.
- Filtering:** On the left, there are filters for "All Items" (selected), Languages, Databases, Middleware, CI/CD, and Other. Below these are filters for "TYPE": Service Class (0), Template (2), Source-to-Image (0), and Installed Operators (0).
- Items:** A grid of items:
 - A red box highlights the "All Items" section.
 - Two cards are shown:
 - ticketmonster-db**: Shows a database icon and a "ticket" button.
 - ticketmonster-monolith**: Shows a monolithic application icon and a "template" button.

Openshift UI Navigation

The screenshot shows the Red Hat OpenShift Container Platform interface. On the left, a sidebar menu includes options like Developer, +Add, Topology, Builds, Advanced, Project Details, Project Access, Metrics, Search, and Events. The main area displays the 'Developer Catalog' for the project 'project-98'. A modal window is open for the 'ticketmonster-monolith' template, which was created on Feb 28, 1:36 pm. The modal contains a large 'Instantiate Template' button and a detailed view of the template's creation information.

Red Hat
OpenShift Container Platform

Developer Catalog

Project: project-98

ticketmonster-monolith

Instantiate Template

ticketmonster-monolith template

CREATED AT

Feb 28, 1:36 pm

All Items

ticket

All Items

ticketmonster-db

ticketmonster-db template

TYPE

Service Class (0)

Template (2)

Source-to-Image (0)

Installed Operators (0)

Openshift UI Navigation

The screenshot shows the Red Hat OpenShift Container Platform interface. The top navigation bar includes the Red Hat logo, 'Red Hat OpenShift Container Platform', and user account information ('user98'). The left sidebar has a 'Developer' section with options like '+Add', 'Topology', 'Builds', 'Advanced' (expanded to show 'Project Details', 'Project Access', 'Metrics', 'Search', and 'Events'), and a 'Create' button. The main content area is titled 'Instantiate Template' and shows a 'Namespace *' dropdown set to 'PR project-98'. To the right, it displays the template name 'ticketmonster-monolith' and its description 'ticketmonster-monolith template'. Below this, it lists the resources to be created: 'DeploymentConfig' and 'Service'. At the bottom right of the content area is a 'Cancel' button.

Openshift UI Navigation

Red Hat OpenShift Container Platform user98

Developer

+Add

Topology

Builds

Advanced >

Project: project-98 Application: all applications

Shortcuts

ticketmonster-m... ticketmonster-db

The screenshot displays the Red Hat OpenShift Container Platform interface. The top navigation bar features the Red Hat logo and the text "Red Hat OpenShift Container Platform". On the right side of the top bar are icons for a plus sign, a question mark, and a user account labeled "user98". Below the top bar is a secondary navigation bar with tabs for "Developer" (selected), "+Add", "Builds", and "Advanced". The main content area is titled "Topology". At the top of this area are dropdown menus for "Project: project-98" and "Application: all applications". To the right of these dropdowns are links for "Shortcuts" and a user icon. The central part of the screen shows two application icons: one for "ticketmonster-m..." and another for "ticketmonster-db", both represented by blue and red circular logos. Below each icon is a small label indicating it is a "DC" (Deployment Config). The overall theme is dark with light-colored text and icons.

Openshift UI Navigation

The screenshot shows the Red Hat OpenShift Container Platform interface. At the top left is the Red Hat logo and the text "Red Hat OpenShift Container Platform". On the far right are user icons for "user98" and a dropdown menu. The left sidebar has a red box around the "Administrator" section, which includes "Home", "Projects", "Search", "Explore", "Events", "Operators", "Workloads", and a "Networking" section with "Services", "Routes" (which is also highlighted with a red box), "Ingresses", and "Network Policies". The main content area shows "Project: project-98" and a "Routes" section with a "Create Route" button and a "Filter by name..." input field. It displays the message "No Routes Found".

Project: project-98 ▾

Routes

Create Route

Filter by name... /

No Routes Found

- Administrator
- Home
- Projects
- Search
- Explore
- Events
- Operators >
- Workloads >
- Networking**
 - Services
 - Routes**
 - Ingresses
 - Network Policies

Openshift UI Navigation

The screenshot shows the OpenShift UI interface. On the left is a dark sidebar menu with the following items:

- Administrator
- Home (highlighted with a red box)
- Projects
- Search (highlighted with a red box)
- Explore
- Events
- Operators >
- Workloads >
- Networking >
- Storage >
- Builds >
- User Management >
- Administration >

The main content area has a header "Project: project-98". Below it is a search bar with a dropdown set to "Service" and a query "app=frontend". A "Create Service" button is located below the search bar. To the right is a "Filter by name..." input field with a search icon.

A table lists two services:

| Name ↑ | Namespace | Labels | Pod Selector | Location |
|------------------------|------------|---|--|---------------------|
| ticketmonster-db | project-98 | app=ticketmonster-db template.openshift.i...=41823047-d237-4... | app=ticketmonster-db, deploymentconfig=ticket monster-db | 172.30.243.100:3306 |
| ticketmonster-monolith | project-98 | app=ticketmonster-monolith template.openshift.i...=6affebd9-1df1-4e... | app=ticketmonster- monolith, deploymentconfig=ticket monster-monolith | 172.30.69.192:8080 |

Step 1: Setup Project

a. Login to OpenShift

b. Create project-##

c. Navigate to your project

The screenshot shows the Red Hat OpenShift Container Platform interface. The top navigation bar includes the Red Hat logo, 'OpenShift Container Platform', and user information for 'user2'. The left sidebar has sections for 'Developer', '+Add', 'Topology' (which is selected and highlighted in blue), 'Builds', and 'Advanced'. The main content area is titled 'Topology' and displays a message: 'No projects exist' and 'Select one of the following options'. A modal window titled 'Create Project' is open, prompting for 'Name *' (with a placeholder value ' ') and other optional fields: 'Display Name' and 'Description'. At the bottom of the modal are 'Cancel' and 'Create' buttons. Below the modal, there are three options: 'From Catalog' (with a book icon), 'Create resources from their YAML or JSON definitions' (with a code editor icon), and 'Browse the catalog to discover database services to add to your application' (with a database icon). A status message at the bottom right of the modal says 'Application will be created.'

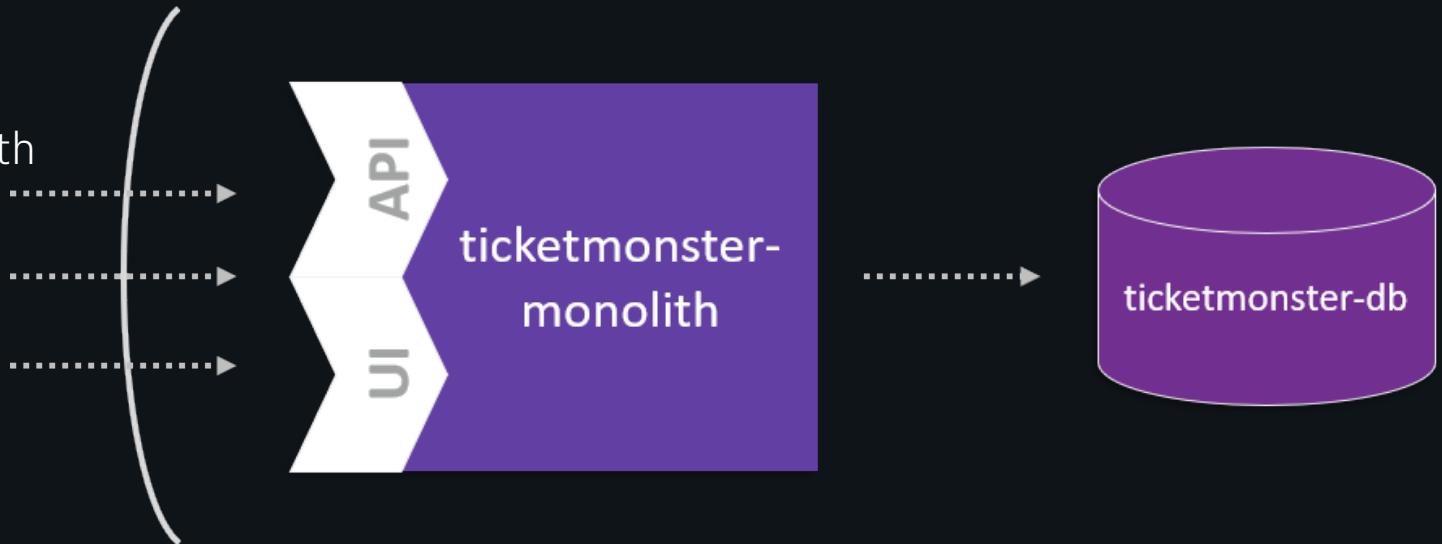
https://github.com/peterhack/Monolith_to_Microservices/tree/master/1_Lift-and-Shift_TicketMonster

Confidential 12



Lab 1: Lift-and-shift TicketMonster

- Overview
 1. Create a MySQL service for monolith
 2. Push application to OpenShift
 3. Bind MySQL service to monolith
- Instructions:
 - Lab: 1_Lift-and-Shift_TicketMonster
- Takeaways
 - Familiarization with OpenShift





Step 1:

Deploy Database for Monolith

a. Click on “Add”

b. Import YAML/JSON

c. Navigate to git repo for ticketmonster-db template

d. Copy RAW YAML

e. Create db template

f. Instantiate db template

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/1_Lift-and-Shift_TicketMonster



Step 2:

Deploy Monolith

a. Click on “Add”

b. Import YAML/JSON

c. Navigate to git repo for **ticketmonster-monolith-template**

d. Copy RAW YAML

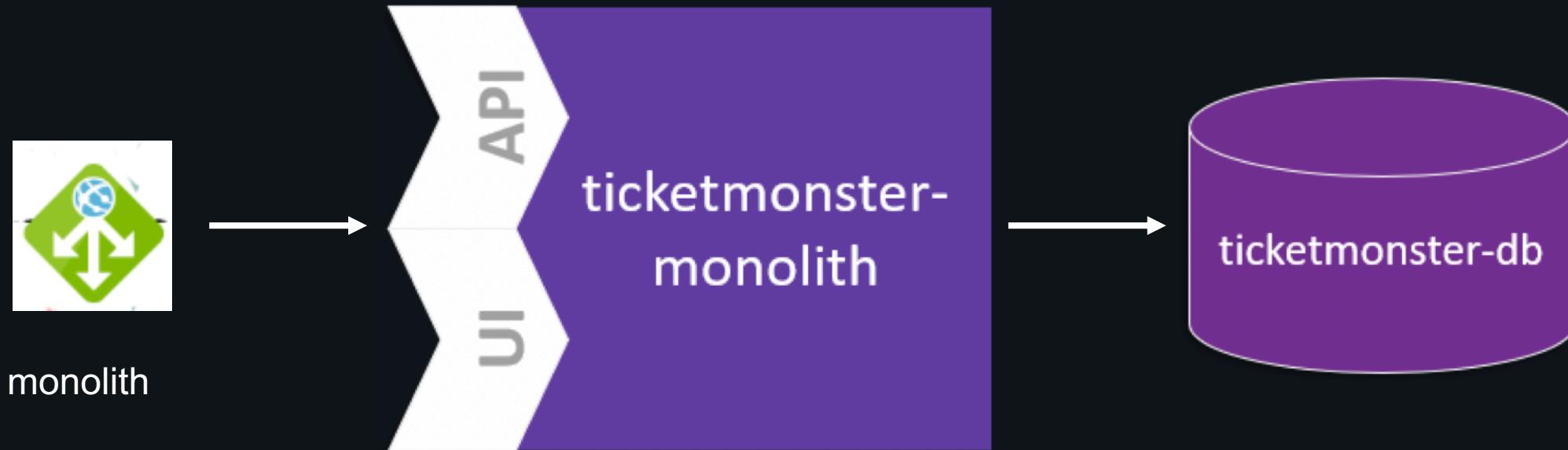
e. Create monolith template

f. Instantiate monolith template

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/1_Lift-and-Shift_TicketMonster



Lab: Lift-and-shift TicketMonster





Step 3:

Create Route to Monolith

- a. Click on “Administrator”**

- b. Networking -> Routes -> Create Route**

- c. Name = monolith**

- d. Service = ticketmonster-monolith**

- e. Target Port = 8080 -> 8080 (TCP)**



Step 4:

Exercise the application

a. In Route, copy “Location”

b. Paste the location url in a browser and exercise the application

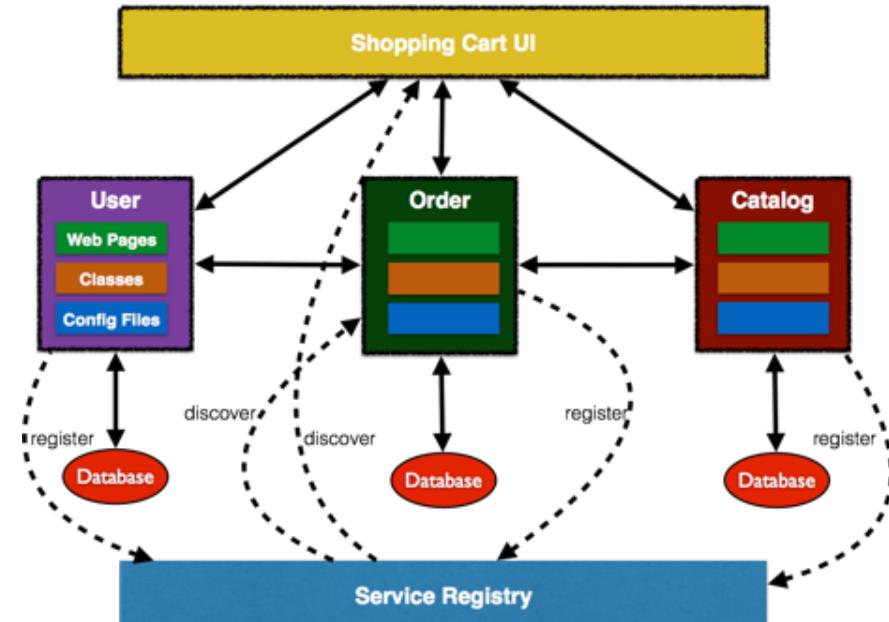
https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/1_Lift-and-Shift_TicketMonster



Extract the UI from the Monolith

A single application consolidating all client interactions for the monolith:

- Application uses the services provided by the monolith and compose them together
 - Use an API gateway or service mesh when application needs to invoke calls to several microservices
- Works as proxy where the UI pages of different components are invoked to show the interface

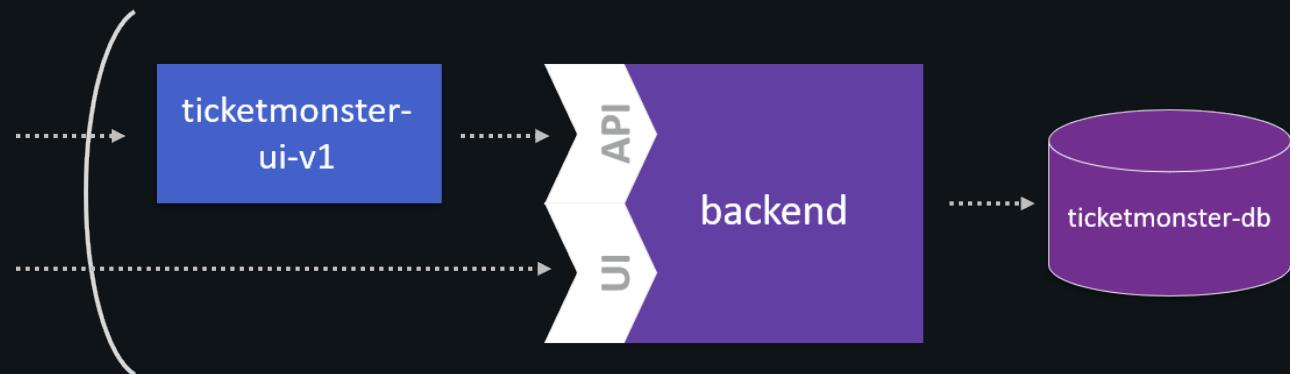




Lab 2: Extract the UI from the Monolith

- Overview
 - 1) Rename 'monolith' to 'backend'
 - 2) Push new UI to OpenShift
 - 3) Switch to tm-ui-v1
- Instructions:
 - Lab: 2_Extract_UI_From_Monolith
- Takeaways
 - Deployment of Microservice

To start breaking up the monolith, a best practice is extracting the user interface from TicketMonster since this helps to decouple the client facing part from the business logic. So, this lab launches the first microservice to separate the user interface from the rest of TicketMonster as depicted below.





Step 1:

Define a new route for the monolith

- a. Click on “Administrator”**

- b. Networking -> Routes -> Create Route**

- c. Name = backend**

- d. Service = ticketmonster-monolith**

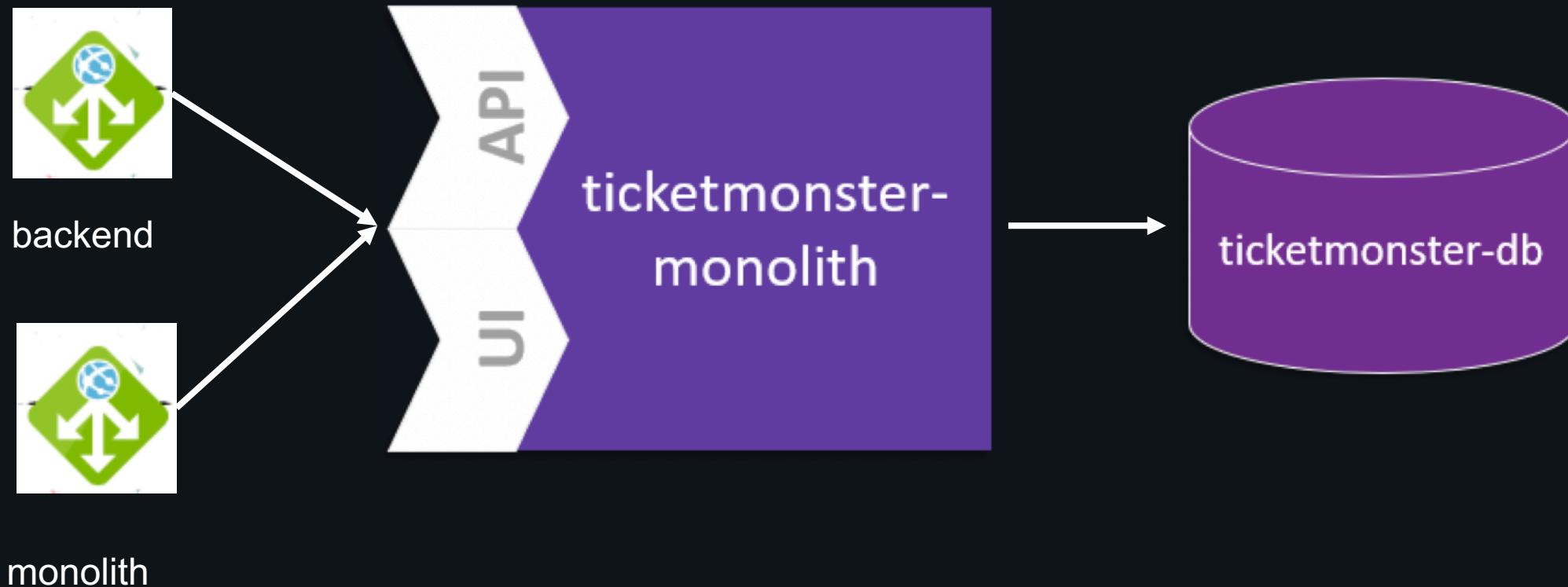
- e. Target Port = 8080 -> 8080 (TCP)**

- f. Copy backend route “host” info in notepad; you will need it later on**

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/1_Lift-and-Shift_TicketMonster



Lab 2: Extract the UI from the Monolith





Step 2:

Decouple UI from the Monolith

a. Click on “Add”

b. Import YAML/JSON

c. Navigate to git repo for **ticketmonster-ui-v1** template

d. Paste RAW YAML into OpenShift

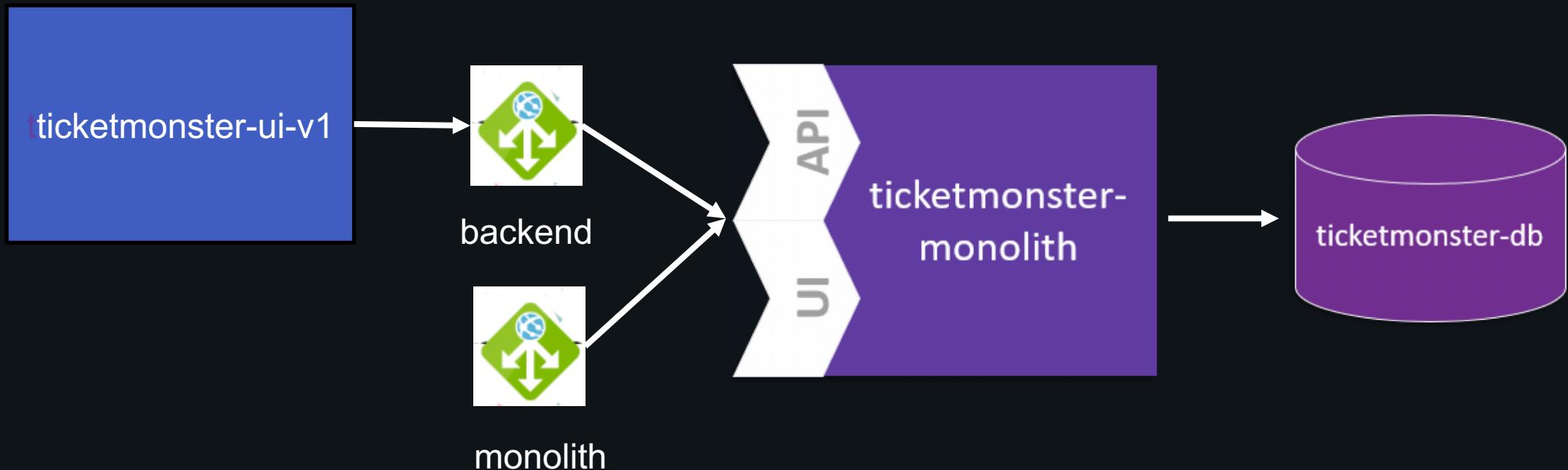
replace backend URL

e. Create & Instantiate monolith template

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/2_Extract_UI_From_Monolith



Lab 2: Extract the UI from the Monolith





Step 1:

Define a new route for the monolith

- a. Click on “Administrator”**

- b. Networking -> Routes -> Create Route**

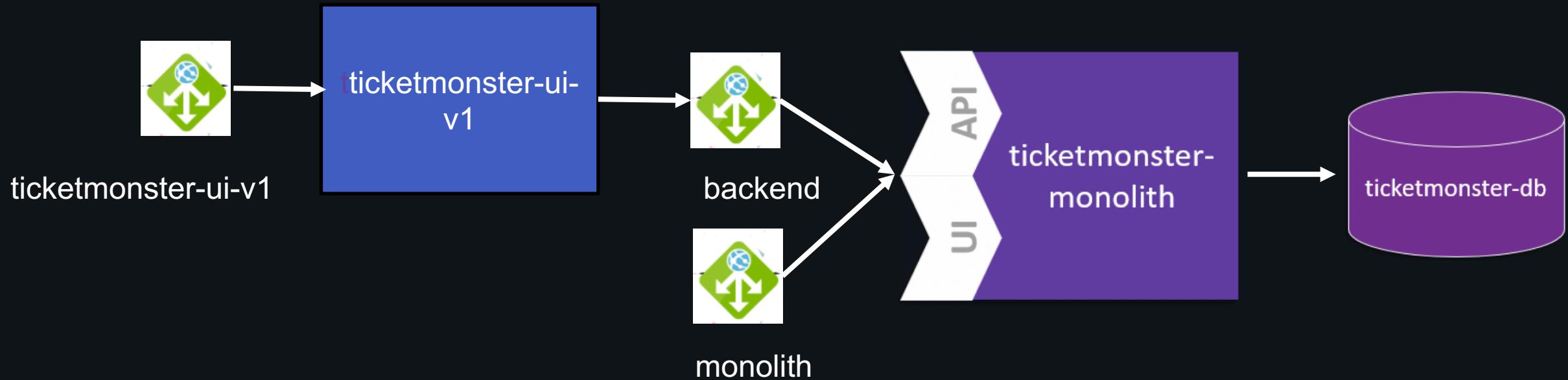
- c. Name = ticketmonster-ui-v1**

- d. Service = ticketmonster-ui-v1**

- e. Target Port = 8080 -> 8080 (TCP)**



Lab 2: Extract the UI from the Monolith





Step 3:

Validate new UI

- a. Select “ticketmonster-UI” URL from the route section and validate that the page states “This UI hits the MONOLITH”

The screenshot shows the homepage of the TicketMonster application. At the top, there's a red header bar with the "TICKETMONSTER" logo. Below it is a grey navigation bar with links: About, Events, Venues, Bookings, Monitor, and Administration. The main content area features a large image of a person performing on stage, with the text "Almost (Mostly) Morrison". To the right of the image, there's a mobile phone icon showing a screenshot of the app's interface. The main text on the page reads "TicketMonster. A JBoss Example." Below this, a paragraph describes the application as an online ticketing demo that helps users learn JBoss technologies. A yellow box highlights the phrase "This UI hits the MONOLITH!". At the bottom, there's a red button labeled "Buy tickets now". A "Fork me on GitHub" badge is visible in the top right corner of the main content area.

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/2_Extract_UI_From_Monolith

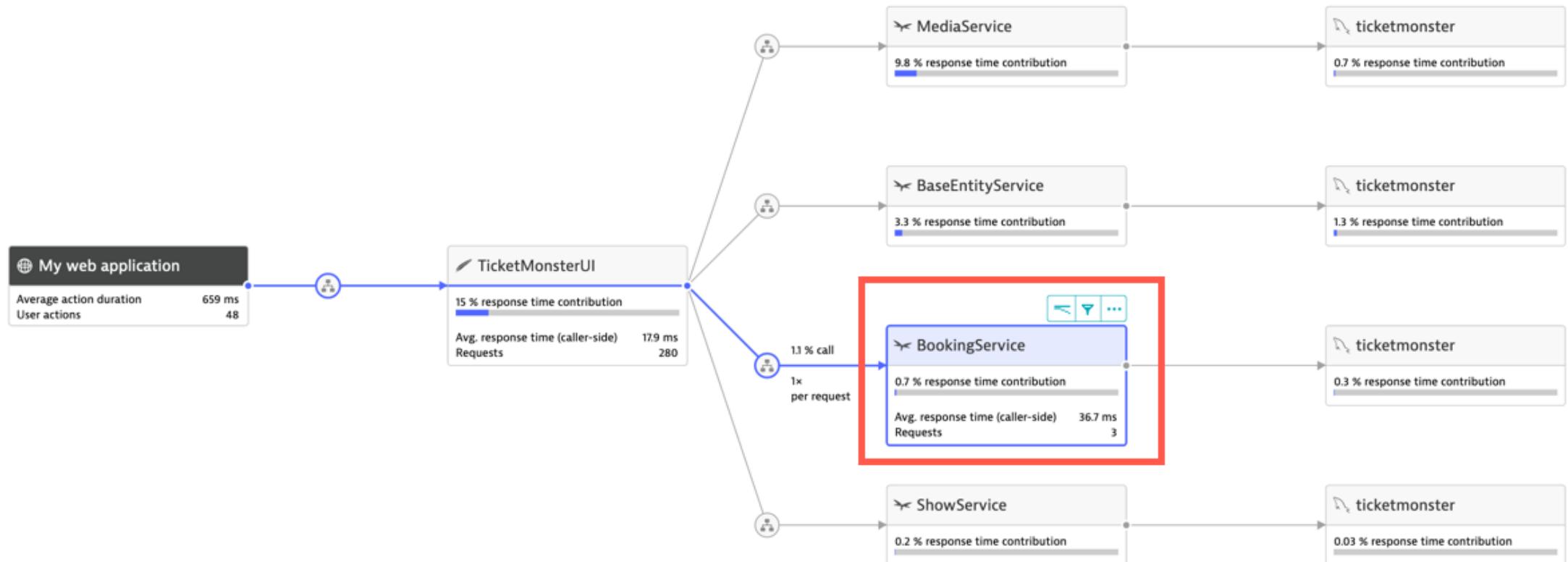


How to identify a Microservice?

- **Problem**: we don't know much about our monolith:
 - Who is depending on us and how are they depending on us?
 - Whom are we depending on and how are we depending on them?
 - What happens within our monolith code base when it gets called?
- **Solution**: Leverage Dynatrace to:
 - Get Dependency Information
 - Detect Service Endpoints, Usage & Behavior
 - Understand Service Flow per Endpoint
 - Finding Entry Points with CPU Sampling Data
 - Define Custom Service Entry Points

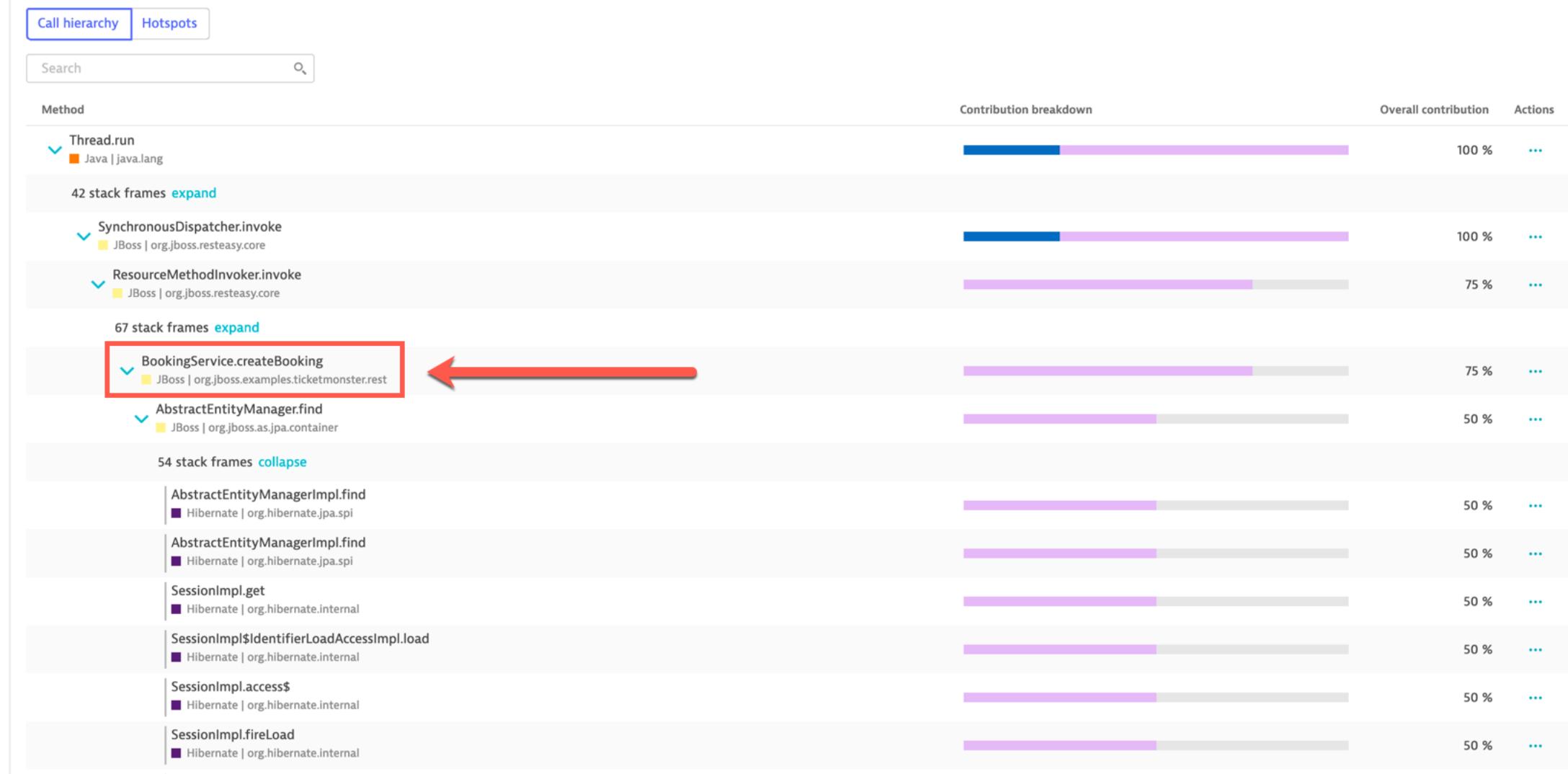


Identify a Microservice – understand dependencies





Identify a Microservice – find the entry point in the code





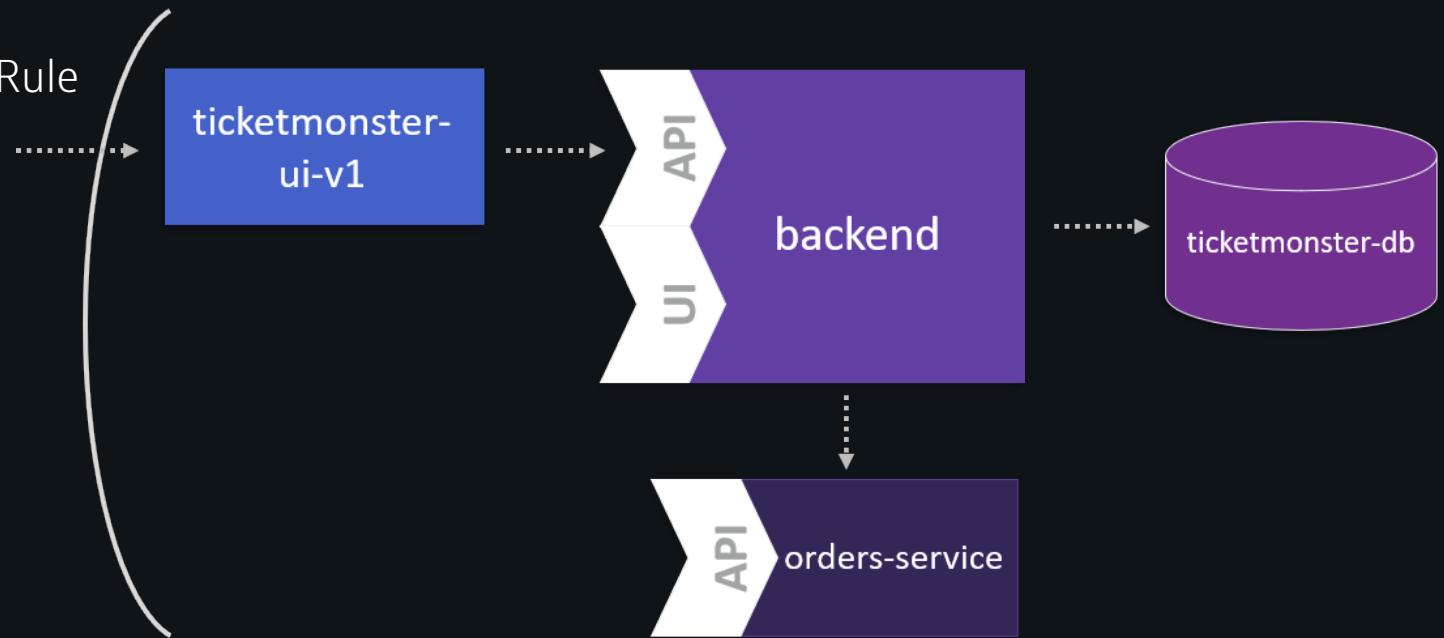
Other considerations to factor in the decision

- Business risk – Is the planned microservice a critical component and does it impact the business in case it fails (conduct a risk assessment)?
- Product roadmap – Which parts of the monolith need to be refactored anyways?
- Business strategy – What is the long-time business strategy of the monolith?
- Technology strategy – Do new technologies impact the further development of the monolith?



Lab 4 : Identify a Microservice - Observability as part of the platform

- Overview
 - 1) Analyze UI and monolith
 - 2) Define Custom Service Detection Rule
- Instructions:
 - Lab: 4_Identify_a_Microservice
- Takeaways
 - Virtually broken the monolith





Step 1:

Define custom service entry point

a. Log into Dynatrace Tenant

b. Go to Settings, Server-side service monitoring, and click on Custom service detection"

c. Click on Define Java service, set name of custom service to "project-xx-orders-service" and click Find entry point

d. Select the process group that contains your entry point "ticketmonster-monolith" and click Continue.



Step 2:

Define custom service entry point

- a. Search for loaded classes and interfaces with name BookingService, select "org.jboss.examples.ticketmonster.rest.BookingService" and click Continue.**

- b. Use Selected Class and click Continue**

- c. Select method with "createBooking" as entry point and click Finish**

- d. Review and select "Save Changes"**



Step 3:

Optional : Restart Pod to activate custom service

a. In Openshift UI, Select Administrator

b. Workloads -> Pods

c. Select Pod “ticketmonster-monolith-xxxx”

d. Action -> Delete Pod



Step 4:

Book a ticket on Ticket Monster

- a. Open the "ticketmonster-ui-v1" in a browser**
- b. Click on Events, Concerts, and on, eg. Rock concert of the decade**
- c. Select Venue, Date, and Order Ticket**
- d. Select section and number of tickets**
- e. Checkout and review booking details**



Step 5:

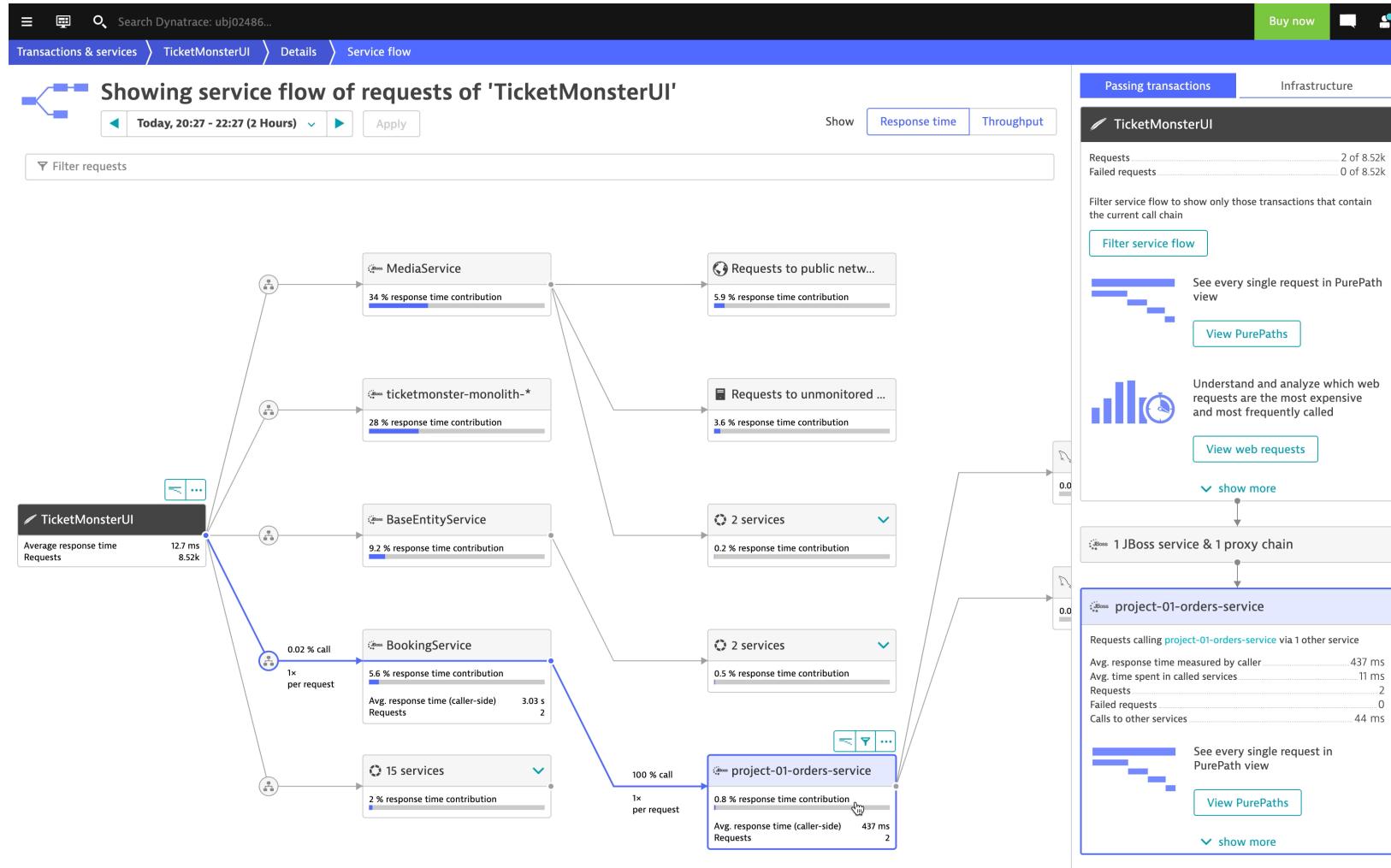
Consider Service Flow in Dynatrace

- a. Log into Dynatrace console and select the Project-## Management Zone
- b. Choose the Transactions & services tab from left menu
- c. Select TicketMonsterUI and click on View service flow



Step 5:

Consider Service Flow in Dynatrace





Identifying the Domain Model of the Microservice

- Data management of a decoupled Microservice:
 - Use an existing API of the monolith to keep data management at the monolith
 - (If no appropriate API is available) create a new lower-level API for the microservice (but still keep the data management at the monolith)
 - Do an ETL (Extract, Transform, Load) from the monolith's database to the microservice' database and keep both in sync
- **Data model** shows how data is stored and entities relate to each other in the persistence layer.
- **Domain model** describes the *behavior* of the solution space of a microservice's domain and tends to focus on use case scenarios



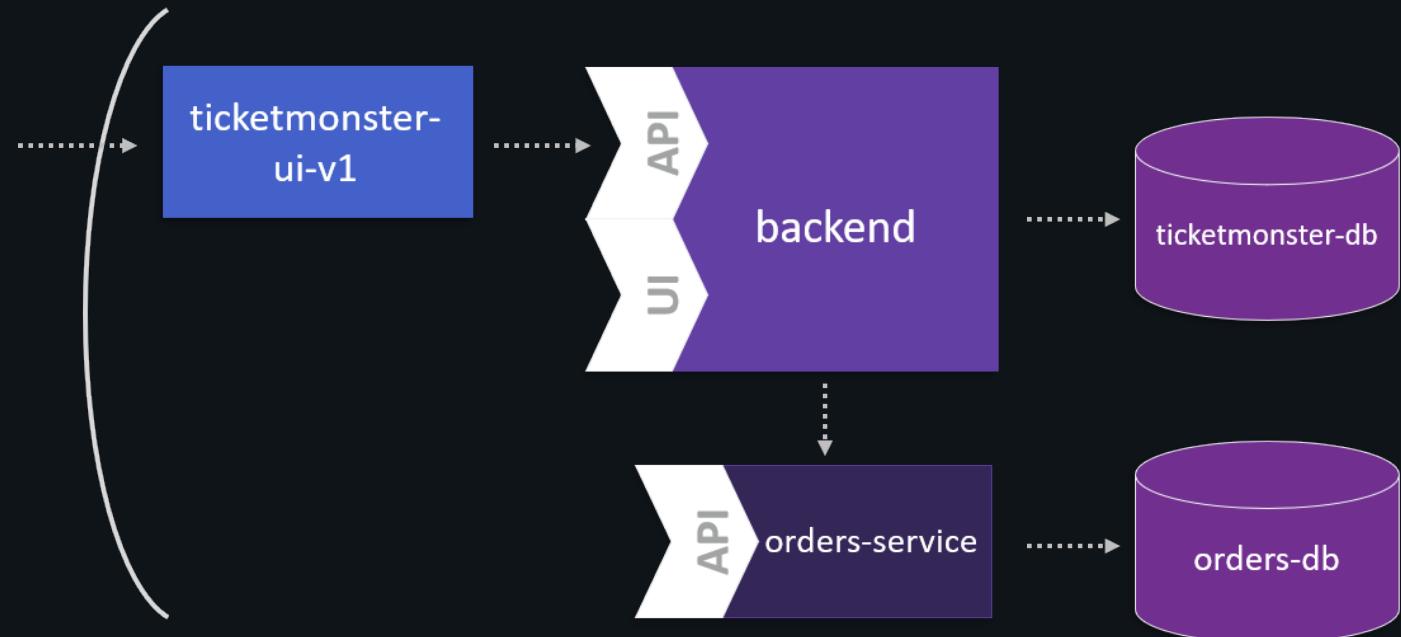
Refactoring your Source Code

- 1) Extract OrdersService from the Monolith
 - Cut out the class containing createBooking method
 - Identify missing abstractions for a successful build
- 2) Strangle the OrdersService around the Monolith
 - Call the OrdersService in the business logic of the backend
- 3) Build the Data and Domain Models for the Microservice
 - Integrate a data virtualization framework into your code base
 - Extend the data model to the domain model by virtualized data views



Lab 5 : The Microservice and its Domain Model

- Overview
 1. Analyze database queries of Microservice
- Instructions:
 - Lab: 5_Domain_Model_of_Microservice
- Takeaways
 - Persistence layer of Microservices





Step 1:

Inspect the Data and Domain Models with Dynatrace

- a. Within Dynatrace Service Flow, follow through custom project-xx-orders-service to mysql db
- b. Click on “View database statements”

Step 2: View database statements



| Statement | Executions | Total executions | Actions |
|---|------------|------------------|---------|
| ■ <code>select section0_id as id1_6_0_, section0_description as descript2_6_0_, section0_name as name3_6_0_, section0_numberOfRows as numberOfRows4_6_0_, section0_rowCapacity as rowCapacity5_6_0_, section0_venue_id as venue_id6_6_0_, venue1_id as id1_11_1_, venue1_city as city2_11_1_, venue1_country as country3_11_1_, venue1_street as street4_11_1_, venue1_capacity as capacity5_11_1_, venue1_description as descript6_11_1_, venue1_mediaItem_id as mediaItem8_11_1_, venue1_name as name7_11_1_, mediaItem2_id as id1_4_2_, mediaItem2_mediaType as update ticket</code> | 0.03 /min | 41 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>insert into ticket</code> | 0.01 /min | 16 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select ticketcate0_id as id1_9_0_, ticketcate0_description as descript2_9_0_ from TicketCategory ticketcate0 where ticketcate0_id=?</code> | 0.01 /min | 11 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select performanc0_id as id1_5_0_, performanc0_date as date2_5_0_, performanc0_show_id as show_id3_5_0_, show1_id as id1_0_1_, show1_event_id as event_id2_0_1_, show1_venue_id as venue_id3_0_1_, event2_id as id1_2_2_, event2_category_id as category4_2_2_, event2_description as descript2_2_2_, event2_mediaItem_id as mediaItem5_2_2_, event2_name as name3_2_2_, ticketPrice3_show_id as show_id4_10_3_, ticketPrice3_id as id1_10_4_, ticketPrice3_price as price2_10_4_, ticketPrice3_section_id as section3_10_4_, ticketPrice3_show_id as select eventcate0_id as id1_3_0_, eventcate0_description as descript2_3_0_ from EventCategory eventcate0 where eventcate0_id=?</code> | 0.01 /min | 8 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select sections0_venue_id as venue_id6_6_0_, sections0_id as id1_6_0_, sections0_id as id1_6_1_, sections0_description as descript2_6_1_, sections0_name as name3_6_1_, sections0_numberOfRows as numberOfRows4_6_1_, sections0_rowCapacity as rowCapacity5_6_1_, sections0_venue_id as venue_id6_6_1_ from Section sections0 where sections0_venue_id=?</code> | 0.01 /min | 8 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select mediaitem0_id as id1_4_0_, mediaitem0_mediaType as mediaType2_4_0_, mediaitem0_url as url3_4_0_ from MediaItem mediaitem0 where mediaitem0_id=?</code> | 0.01 /min | 8 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select performanc0_show_id as show_id3_5_0_, performanc0_id as id1_5_0_, performanc0_id as id1_5_1_, performanc0_date as date2_5_1_, performanc0_show_id as show_id3_5_1_ from Performance performanc0 where performanc0_show_id=? order by performanc0_date</code> | 0.01 /min | 8 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select sectionall0_id as id1_7_, sectionall0_allocated as allocate2_7_, sectionall0_occupiedCount as occupied3_7_, sectionall0_performance_id as performance5_7_, sectionall0_section_id as section6_7_, sectionall0_version as version4_7_ from SectionAllocation sectionall0 where sectionall0_performance_id=? and sectionall0_section_id=?</code> | 0 /min | 6 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select id from SectionAllocation where id=? and version=? for update</code> | 0 /min | 6 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>update sectionallocation</code> | 0 /min | 6 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>insert into booking</code> | 0 /min | 6 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>sql commit</code> | 0 /min | 6 ↗️ ↘️ ⚡ | ▼ |
| ■ <code>select ticketPrice0_id as id1_10_, ticketPrice0_price as price2_10_, ticketPrice0_section_id as section3_10_, ticketPrice0_show_id as show_id4_10_, ticketPrice0_ticketCategory_id as ticketCa5_10_ from TicketPrice ticketPrice0 where ticketPrice0_id in (?)</code> | 0 /min | 4 ↗️ ↘️ ⚡ | ▼ |

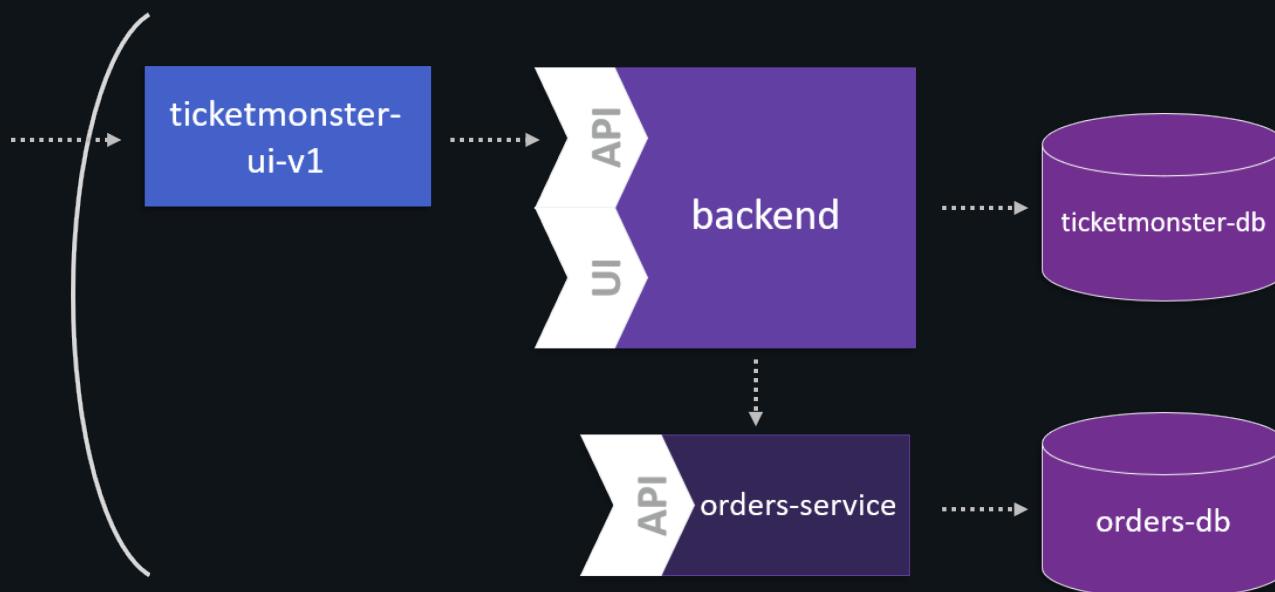


Lab 6 : Deploy the Microservice

- Overview
 1. Deploy a new backend version
 2. Create a MySQL service instance for microservice
 3. Build microservice and push it to OpenShift
 4. Bind MySQL service instance to microservice
 5. Re-deploy UI to use new backend version
- Instructions:
 - Lab: 6_Deploy_the_Microservice
- Takeaways
 - Microservice for a specific bounded context

Based on the result of the previous labs, you identified the microservice **OrdersService** that has its own code repository and defines its own domain model. To launch this service, it is not recommended to directly route traffic to this new service since we cannot fully ensure that it works as supposed. For this reason, we strangle the microservice around the monolith. In other words, all incoming requests will still be intercepted by the backend service, which forwards synthetic or live traffic to **OrdersService**.

In this lab you'll use feature flags and OpenShift routing mechanism to smoothly incorporate the new microservice into the monolith. The final state of this lab is shown below:





Step 1:

Create the new data store for the Orders microservice

a. Select “Add”

b. Import YAML/JSON

c. Navigate to git repo for ticketmonster orders-db template

d. Paste RAW YAML into OpenShift

e. Create and Instantiate



Step 2:

Setup the database

a. Administrator -> Workloads -> Pod

b. Select the “orders-db” pod

c. Select the “Terminal” tab

d. In the terminal type:

a. cd~

b. pwd

*Should now be in directory /var/lib/mysql



Step 2: Setup the database

```
curl -o 0_ordersdb-schema.sql https://raw.githubusercontent.com/dynatrace-innovationlab/monolith-to-microservice-openshift/master/orders-service/src/main/resources/db/migration/0_ordersdb-schema.sql

curl -o 1_ordersdb-data.sql https://raw.githubusercontent.com/dynatrace-innovationlab/monolith-to-microservice-openshift/master/orders-service/src/main/resources/db/migration/1_ordersdb-data.sql

mysql -u root orders < 0_ordersdb-schema.sql

mysql -u root orders < 1_ordersdb-data.sql

mysql -u root

GRANT ALL ON orders.* TO 'ticket'@'%';

show databases;

use orders

show tables;

select * from id_generator;
```



Step 4:

Deploy the Orders microservice

a. Developer -> Add

b. Navigate to git repo for ticketmonster-orders-service template

c. Copy RAW YAML

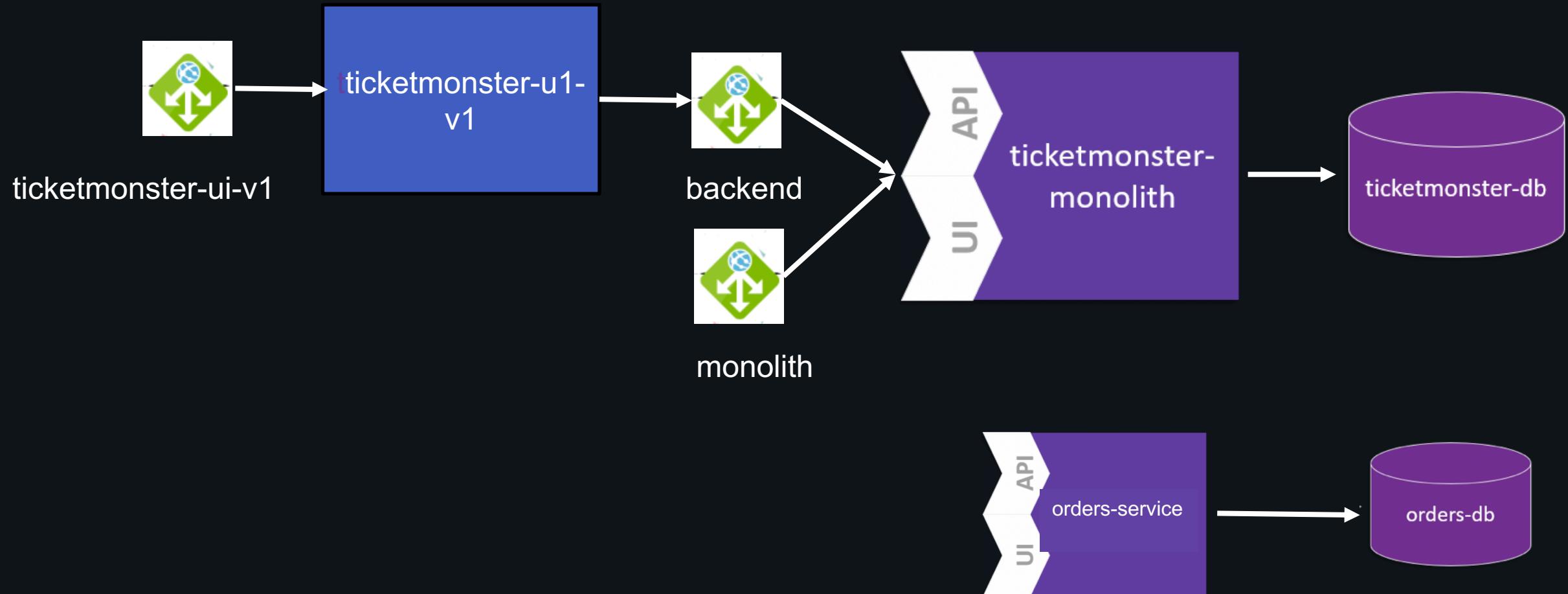
d. Paste RAW YAML into OpenShift

e. Create and Instantiate

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/6_Deploy_the_Microservice



Lab 6 : Deploy the microservice





Step 5:

Deploy ticketmonster-backend-V2

a. Developer -> Add

b. Navigate to git repo for ticketmonster-backend-v2 template

c. Copy RAW YAML

d. Paste RAW YAML into OpenShift

e. Create and Instantiate

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/6_Deploy_the_Microservice



Step 6:

Create route for ticketmonster-backend-V2

a. Administrator -> Networking - Routes

b. Select “Create Route”

c. Name: backend-v2

d. Service: ticketmonster-backend-v2

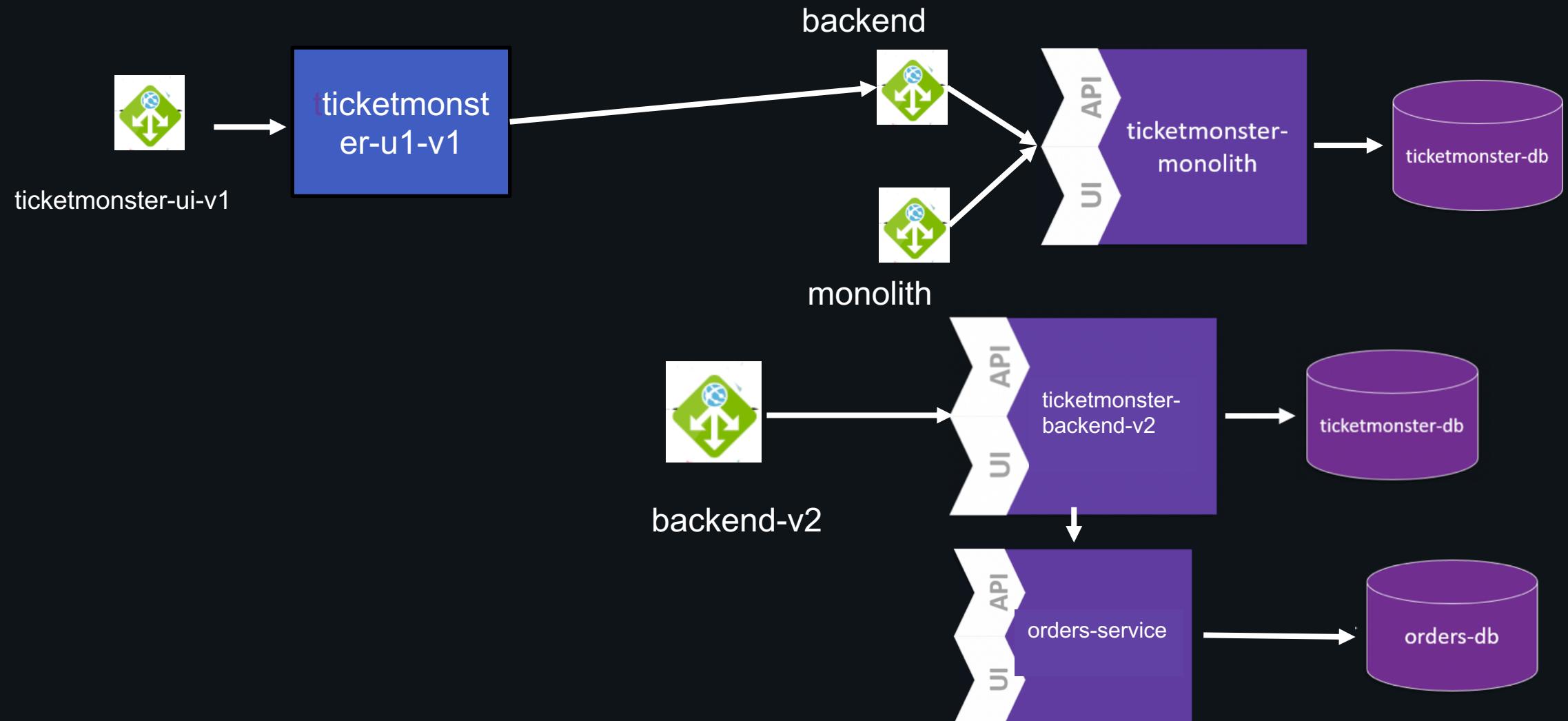
e. Target Port: 8080 -> 8080 (TCP)

f. Select “Create”

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/6_Deploy_the_Microservice



Lab: Identify the microservice





Step 7:

Split route to backend

a. Split the traffic to the backend

Can be done from the CLI :

```
oc set route-backends backend ticketmonster-monolith=50 ticketmonster-backend-v2=50 -n project-xx
```

Or can also be done by editing the yaml in the console:

- In Administrator mode, go in the Routes definitions.
- Select the backend route
- Click Edit
- Add the following yaml after the spec:to , also change the weight in the “to” from 0 to 50

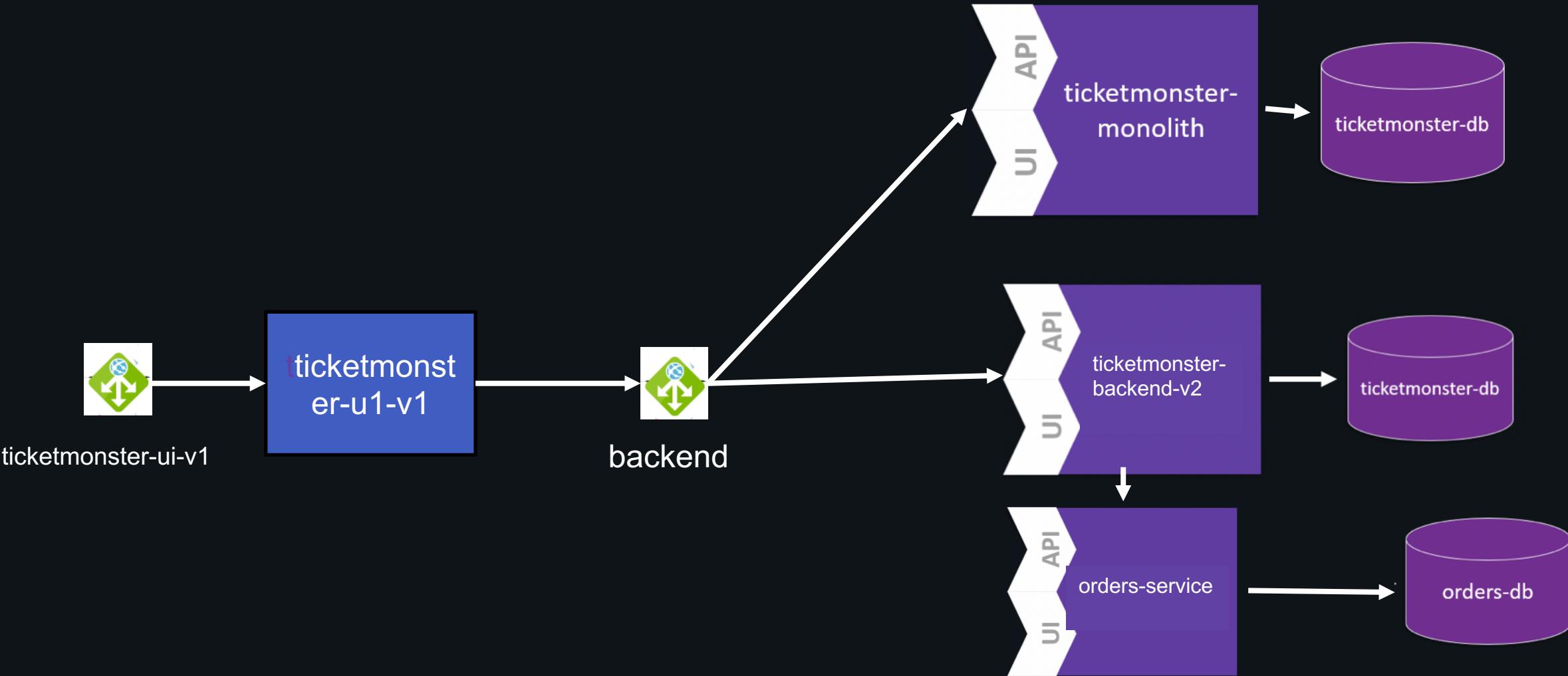
alternateBackends:

- kind: Service
name: ticketmonster-backend-v2
weight: 50

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/6_Deploy_the_Microservice



Lab: Identify the microservice





Step 8:

Verify Service in Dynatrace

We can verify the service flow in Dynatrace.

From the left menu, choose the Transaction & services tab.

Select service TicketMonsterUi.

Click on View service flow.

Finally, you see the service flow containing the microservice orders-service.

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/6_Deploy_the_Microservice



Step 8:

Verify Service in Dynatrace

The screenshot shows the Dynatrace interface for monitoring microservices. On the left, a sidebar lists various monitoring categories like Dashboards & reports, Transactions & services, and Applications. The main area displays a service flow for the 'OrdersService'. The flow starts with the 'OrdersService' itself, which has an average response time of 987 ms and 6 requests. Two arrows point from this service to two external services: 'ticketmonster' and 'orders'. 'ticketmonster' is highlighted with a blue border and shows a 46% response time contribution. Below the flow diagram, there's a note: 'No service selected. Select any service in the service flow to get more details and perform deeper analysis.' To the right of the flow, there are two sections: 'Passing transactions' and 'Infrastructure'. The 'Passing transactions' section for 'OrdersService' shows metrics: Avg. response time (987 ms), Avg. time spent in called services (489 ms), Requests (6), Failed requests (0), and Calls to other services (89). It also includes a callout for 'View PurePaths'. The 'Infrastructure' section is currently inactive.

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/6_Deploy_the_Microservice



Summary

- Limitation to Monoliths
 - Agility – Rebuilding the whole application takes a decent amount of time
 - Scalability – Scaling a monolith happens in both directions: vertically as well as horizontally - causing unused resources
 - DevOps Cycle – Continuous delivery (high frequency of deployments) fails due to high build time
- Limitations of Monoliths have given rise to Microservices
 - Agility - Scope changes can be done in one microservice - other micro services are not impacted from these changes
 - Scalability - Individual components can scale as needed
 - DevOps Cycle - Since each component operates independently, continuous delivery cycle reduces
- Dynatrace can be leveraged in this Monolith to Microservices Journey to:
 - Get Dependency Information
 - Detect Service Endpoints, Usage & Behavior
 - Understand Service Flow per Endpoint
 - Finding Entry Points with CPU Sampling Data
 - Define Custom Service Entry Points



dynatrace.com