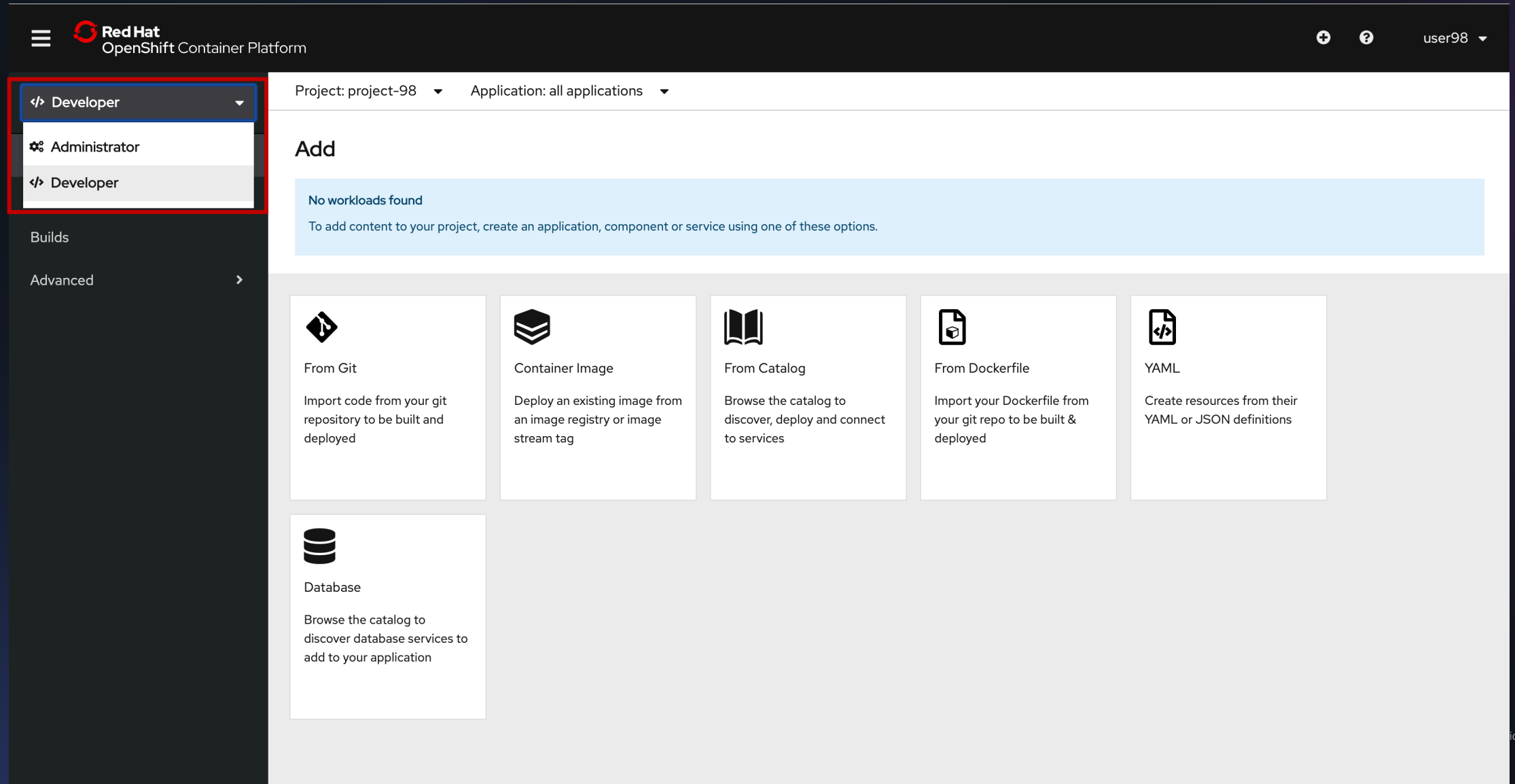
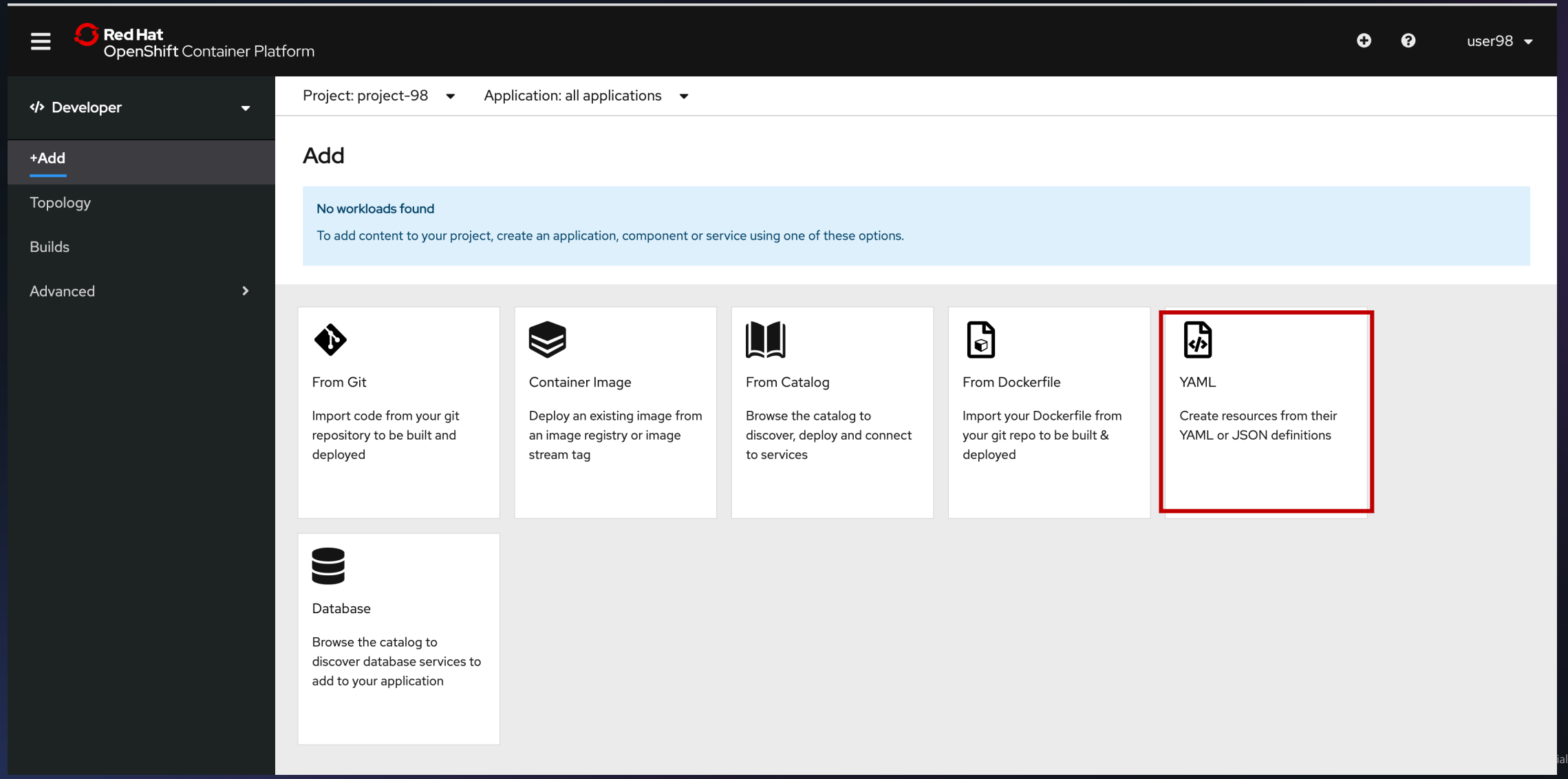
Hands-on with TicketMonster

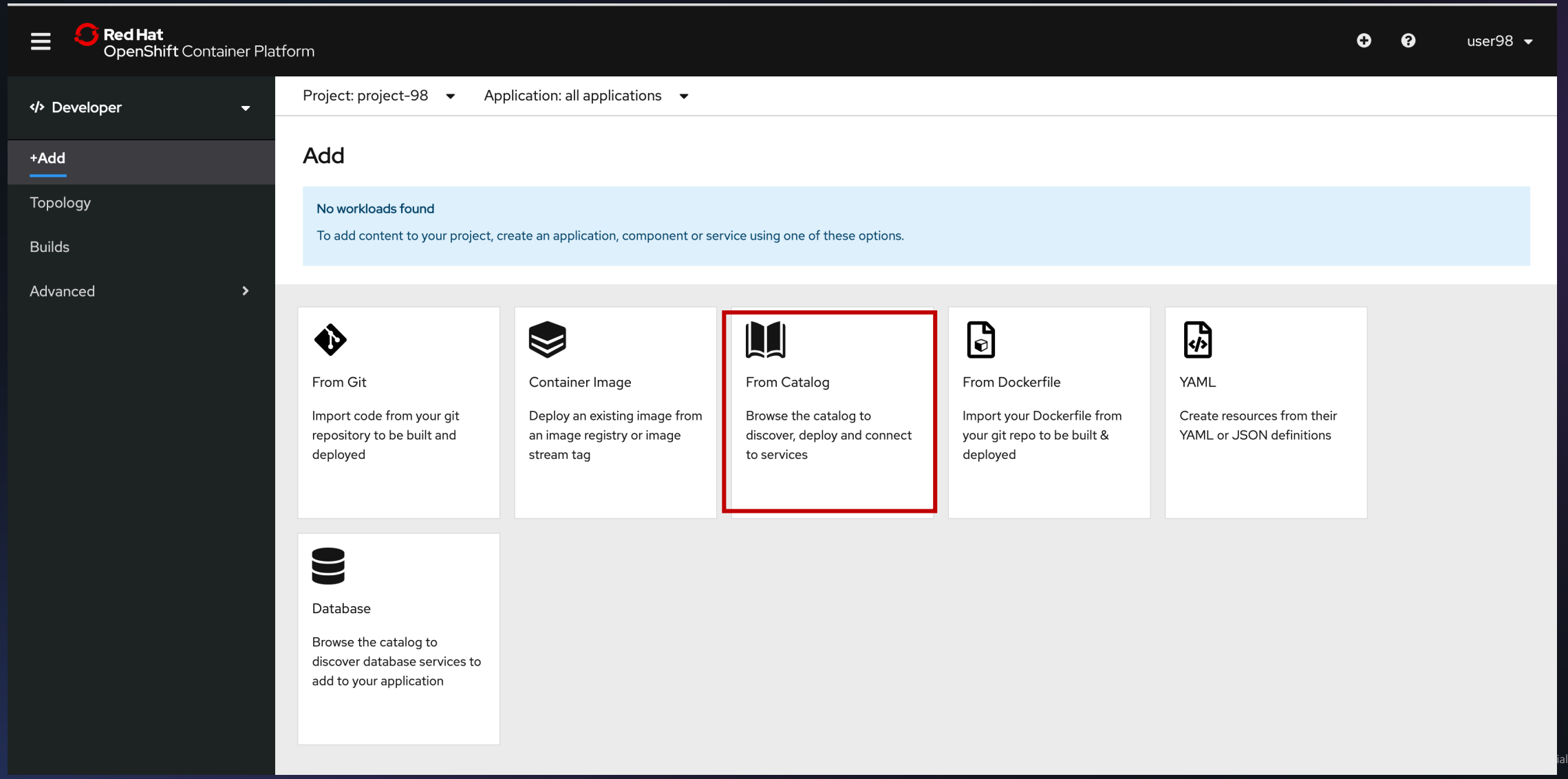
Openshift UI Navigation



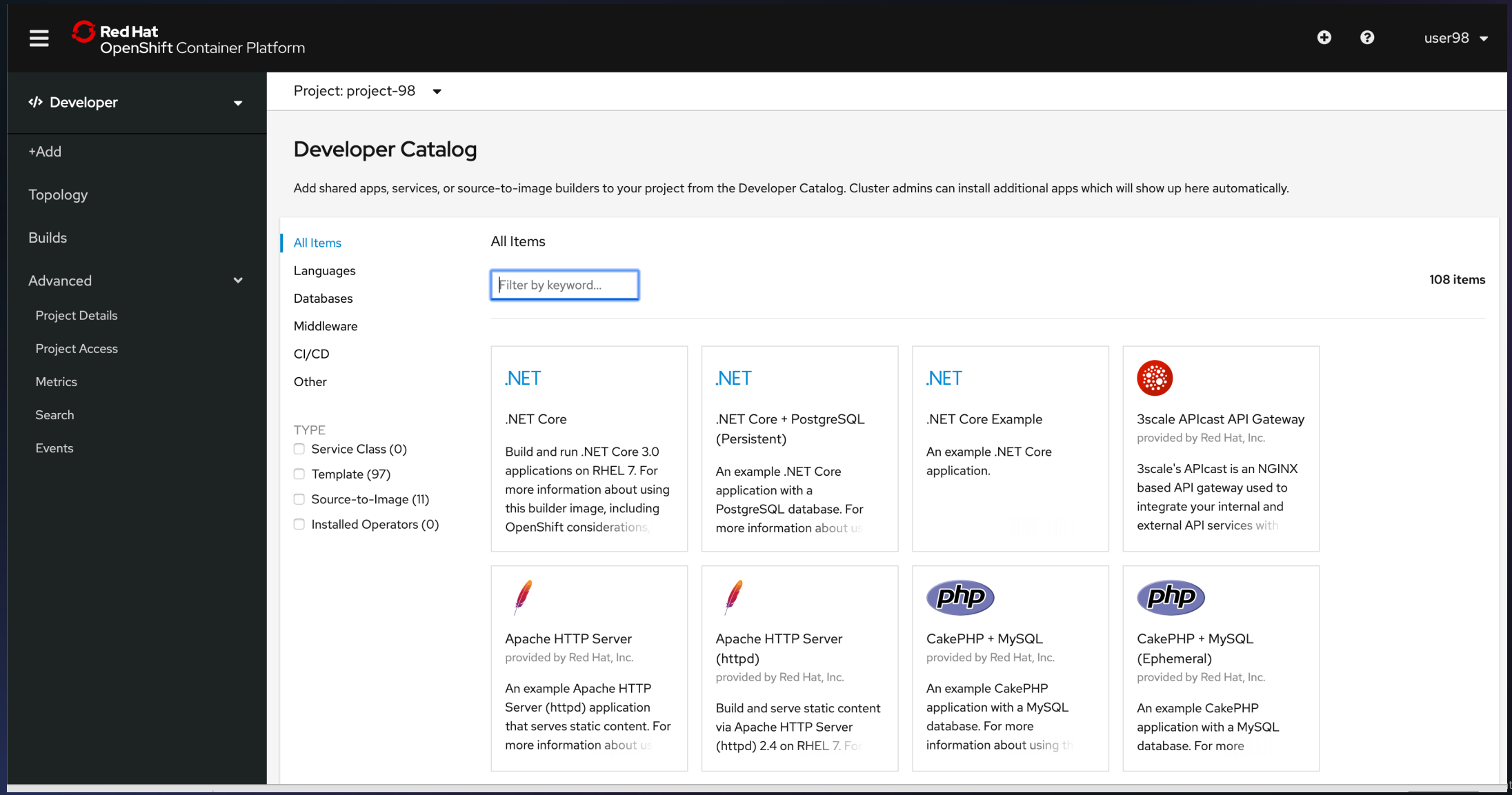
Openshift UI Navigation



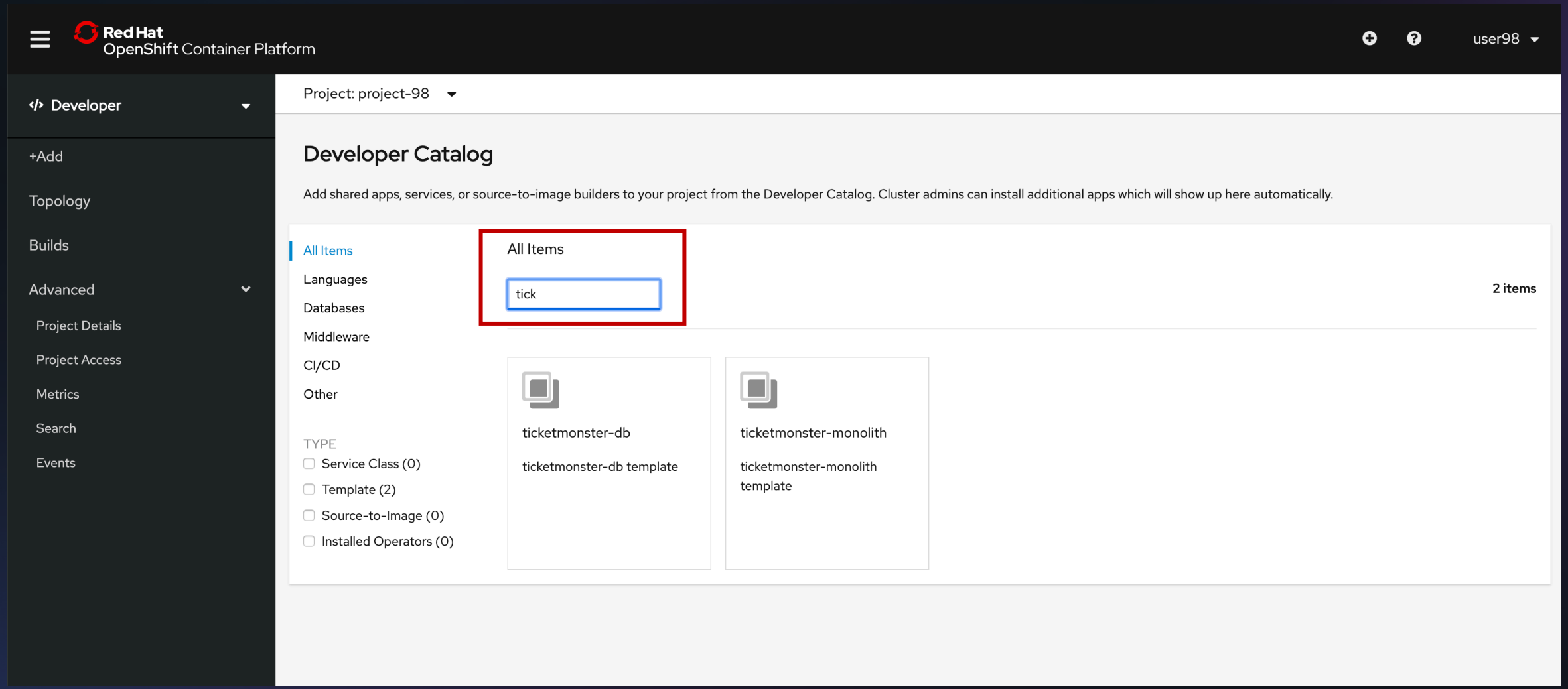
Openshift UI Navigation



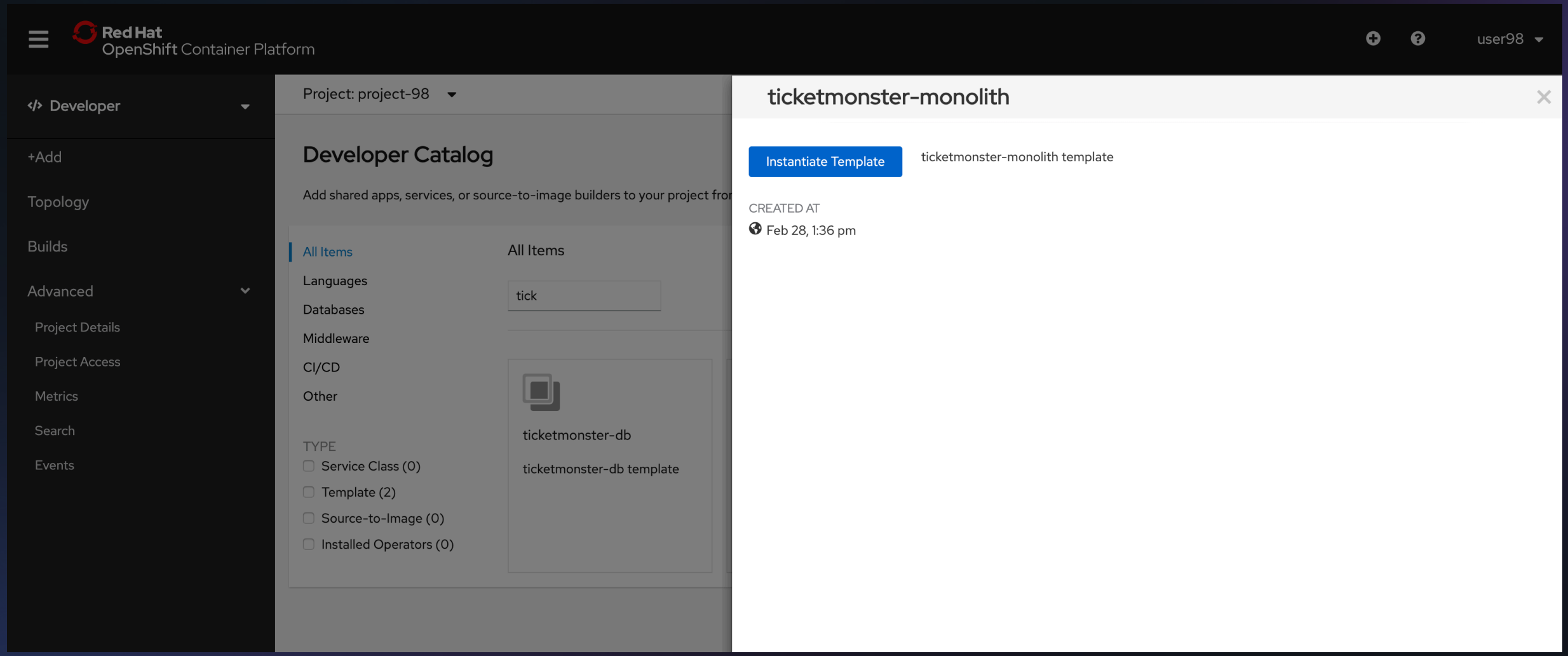
Openshift UI Navigation



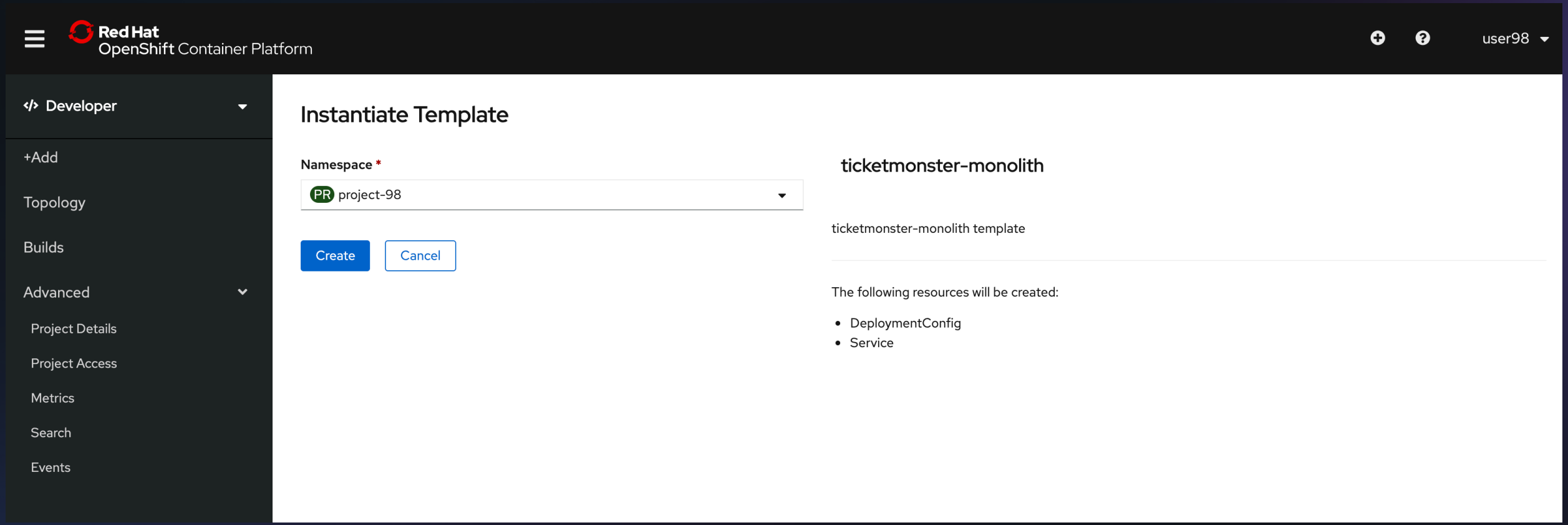
Openshift UI Navigation



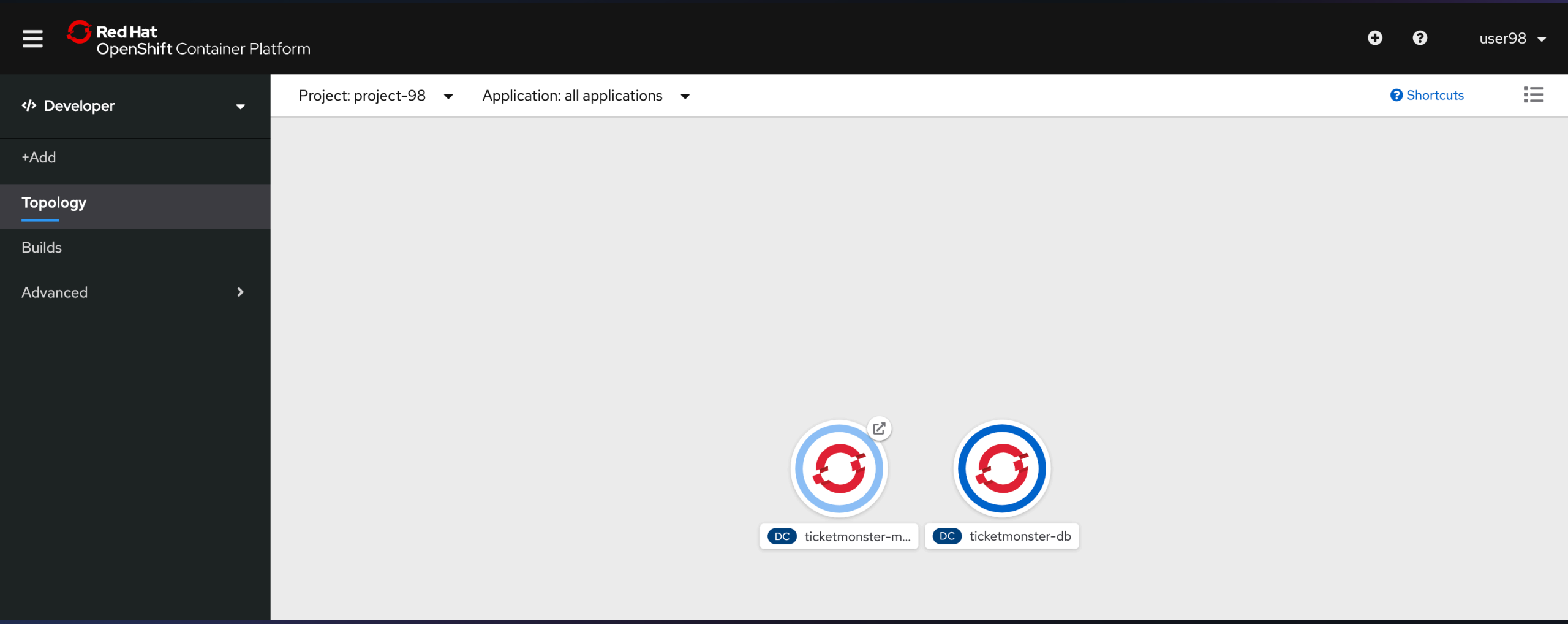
Openshift UI Navigation



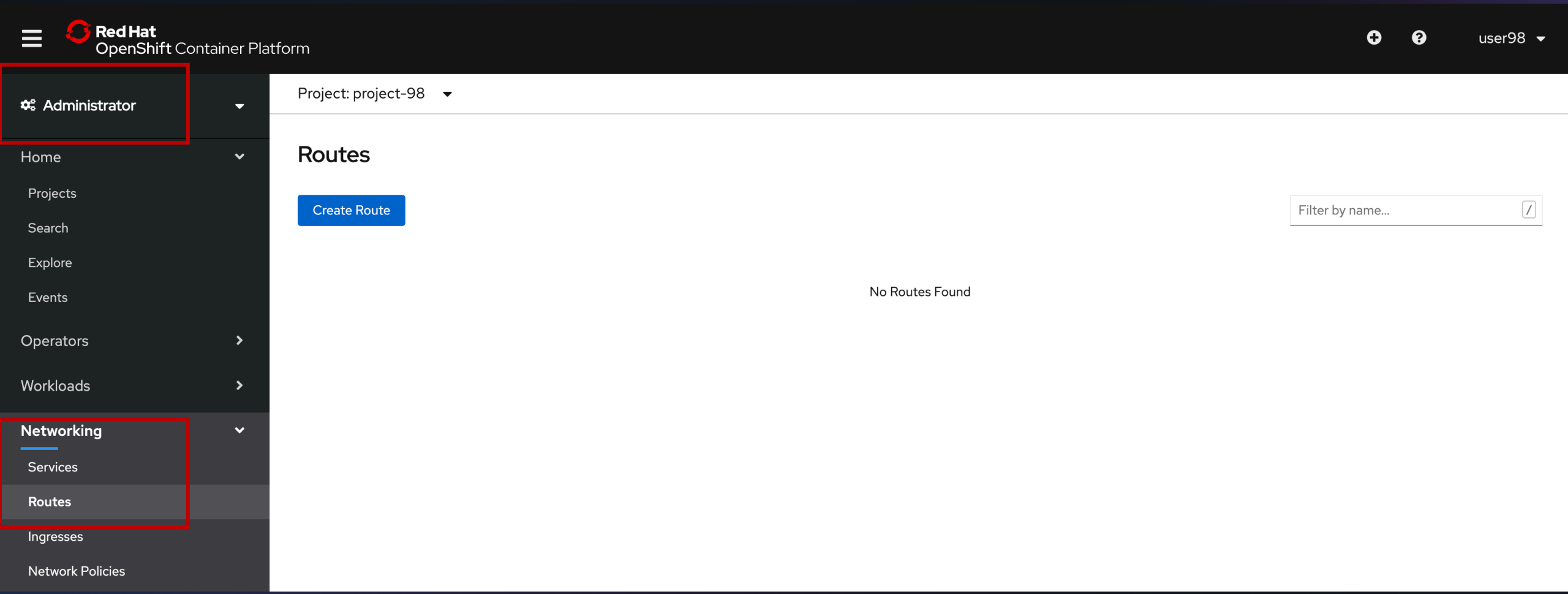
Openshift UI Navigation



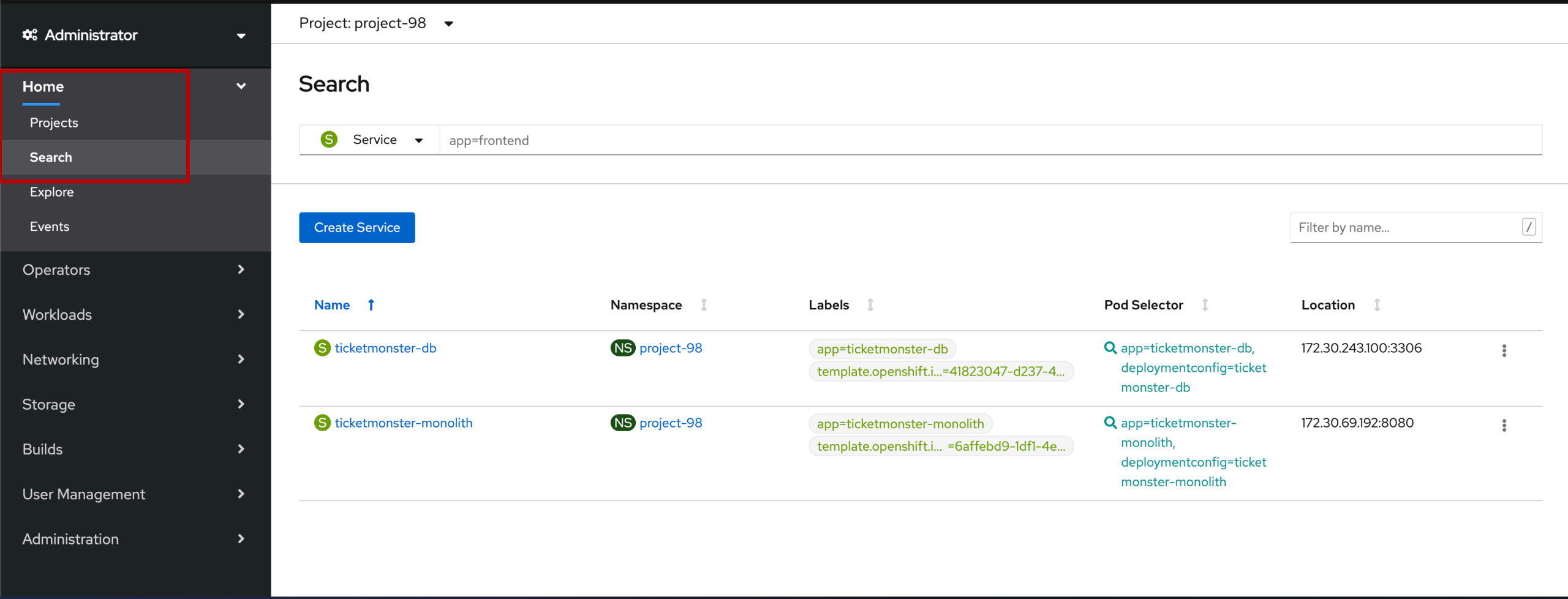
Openshift UI Navigation



Openshift UI Navigation



Openshift UI Navigation



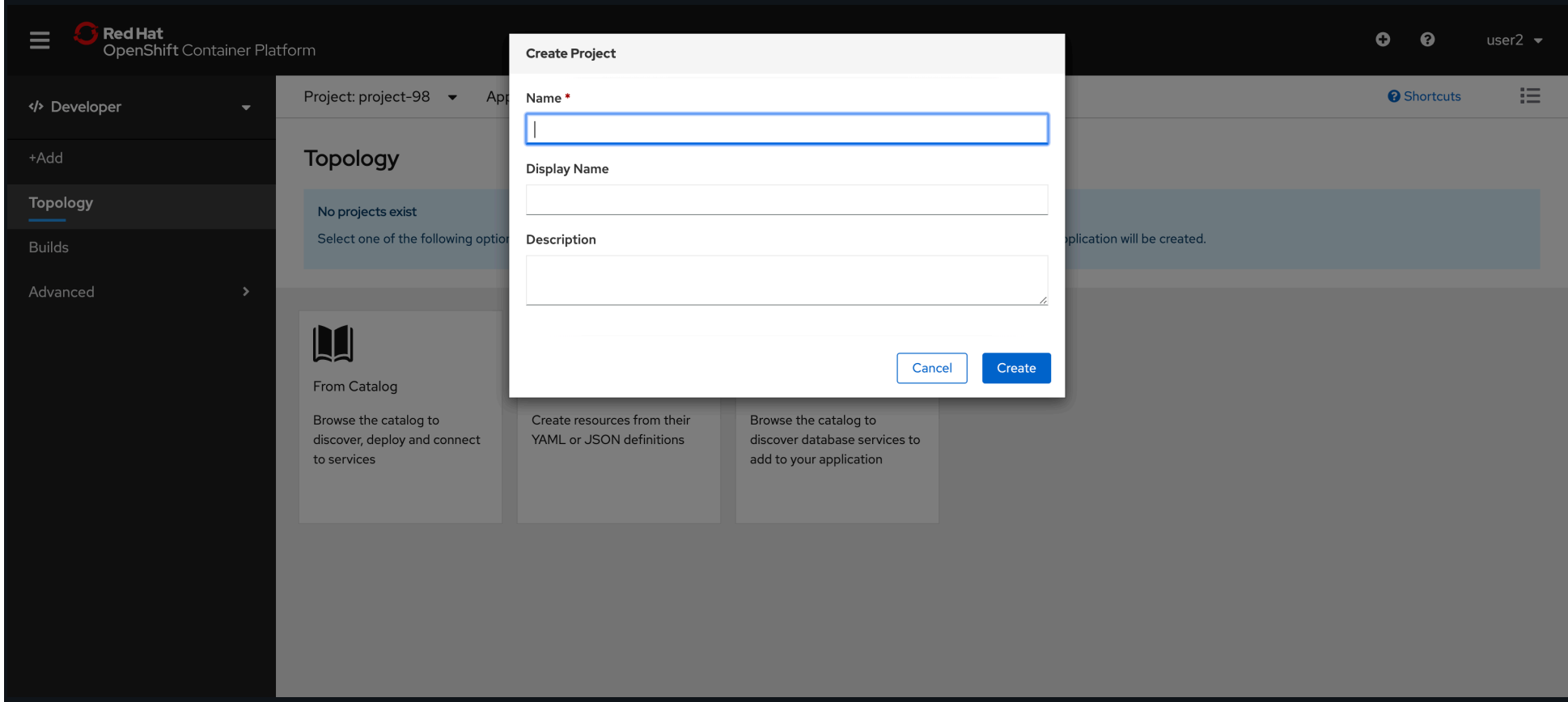
Step 1:

Setup Project

a. **Login to OpenShift**

b. **Create project-##**

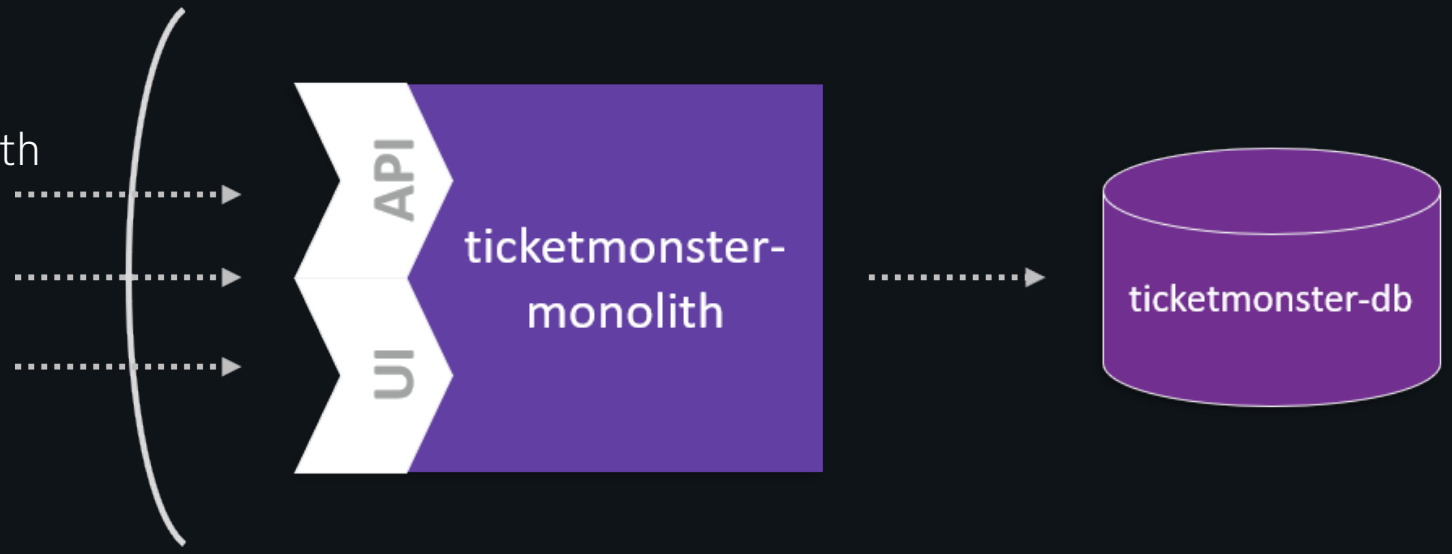
c. **Navigate to your project**



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

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

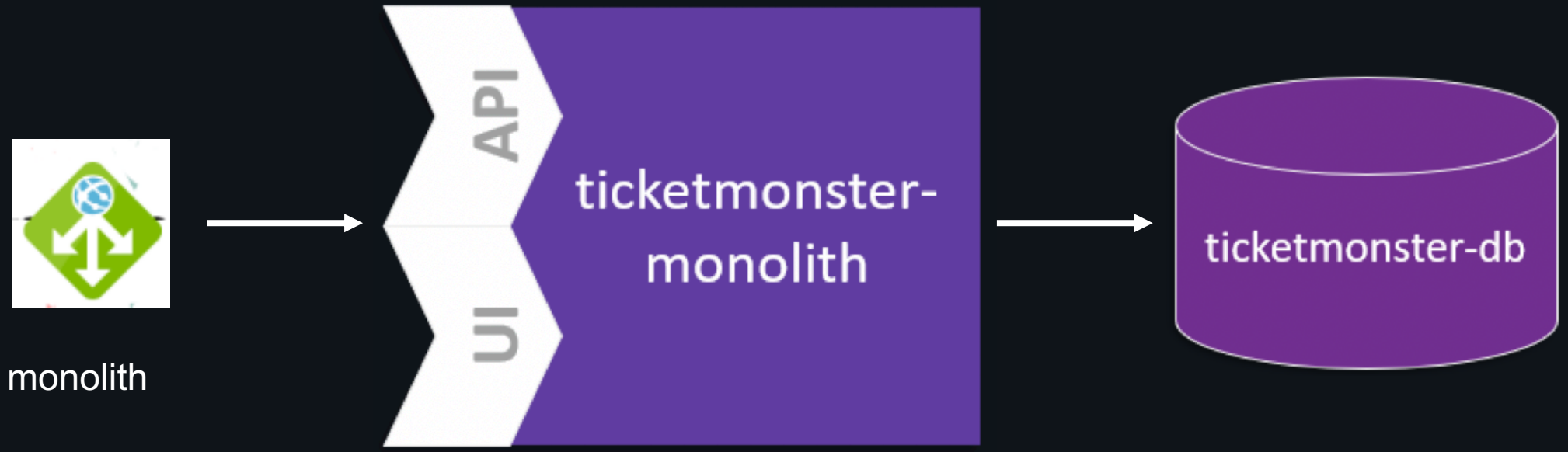
d. Copy RAW YAML

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)

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

Step 4:

Exercise the application

a. In Route, copy “Location”

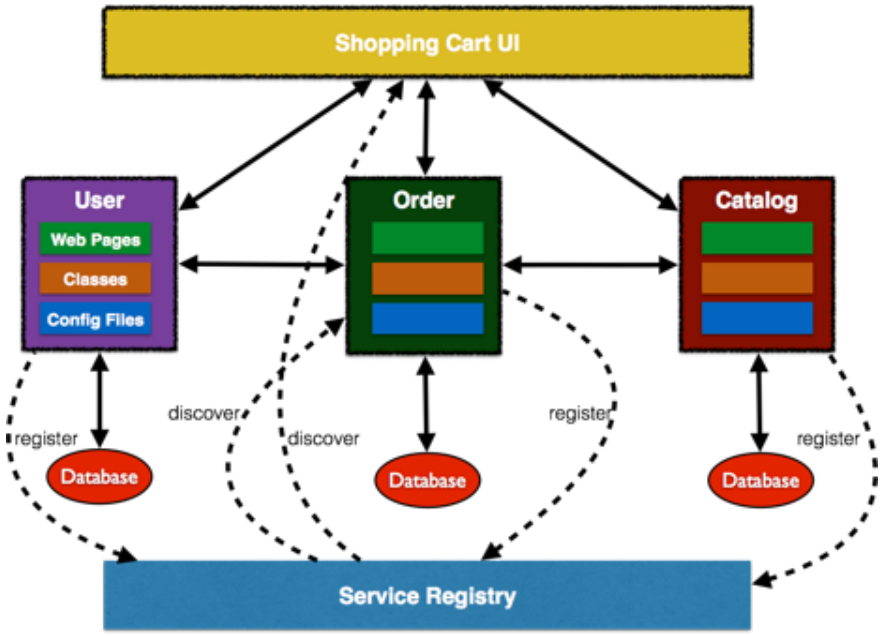
b. Paste the location url in a browser and exercise the application

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

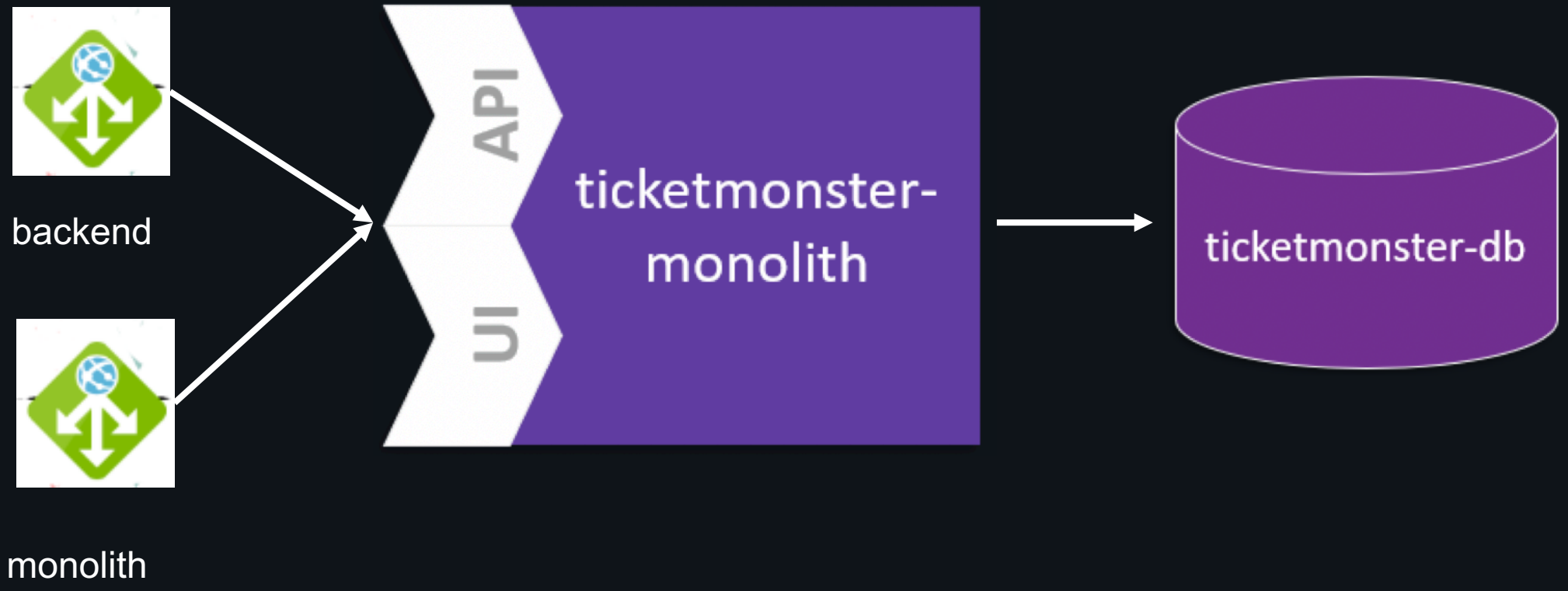
d. Service = ticketmonster-monolith

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

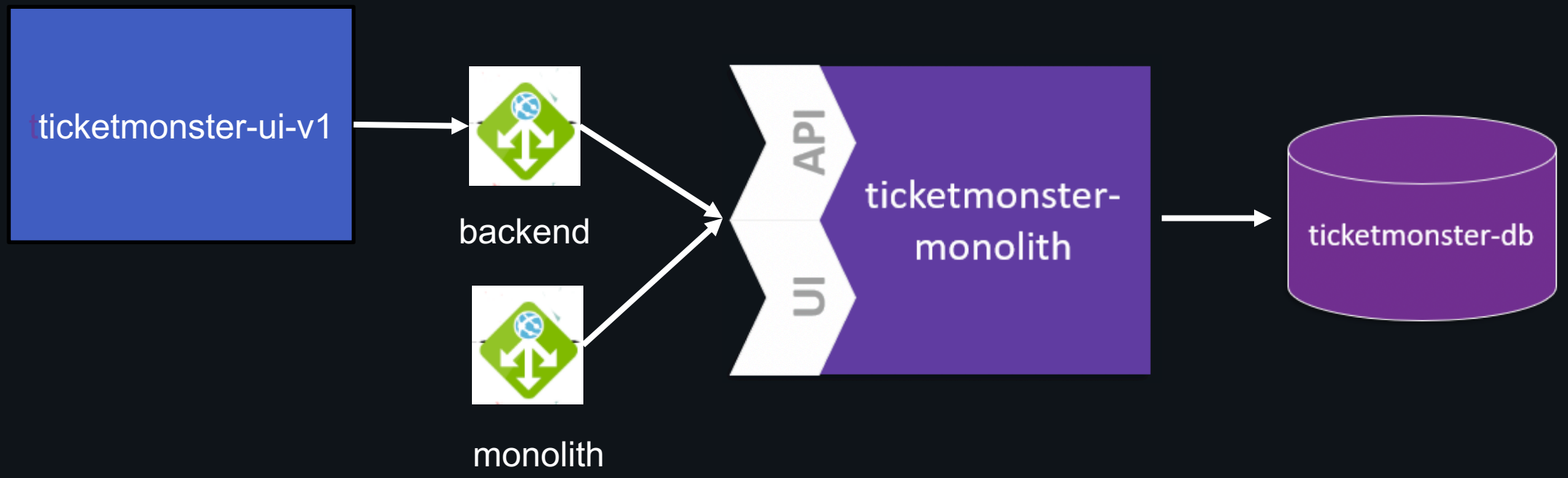
d. Paste RAW YAML into OpenShift

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”

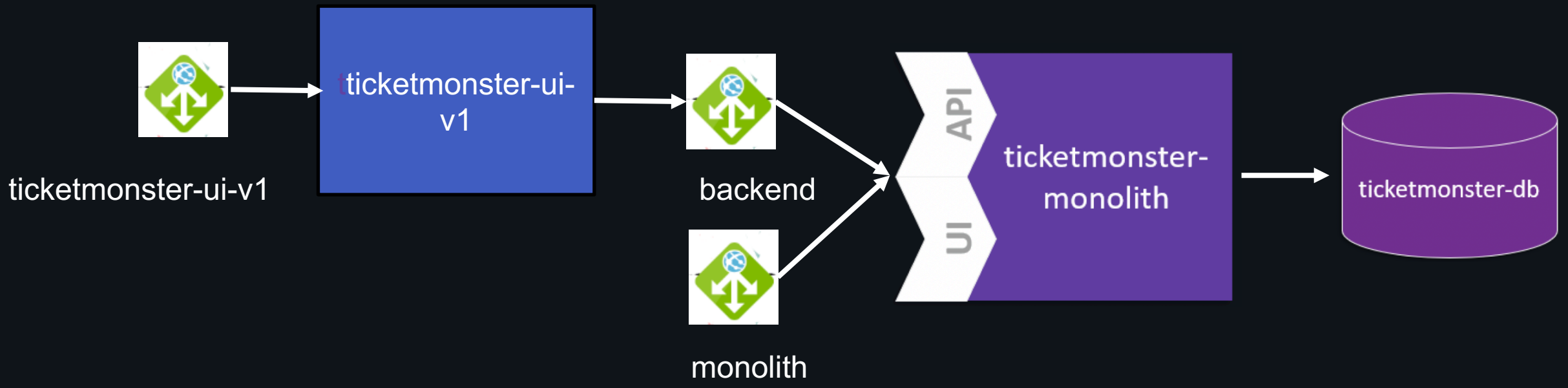
b. Networking -> Routes -> Create Route

c. Name = ticketmonster-ui-v1

d. Service = ticketmonster-ui-v1

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

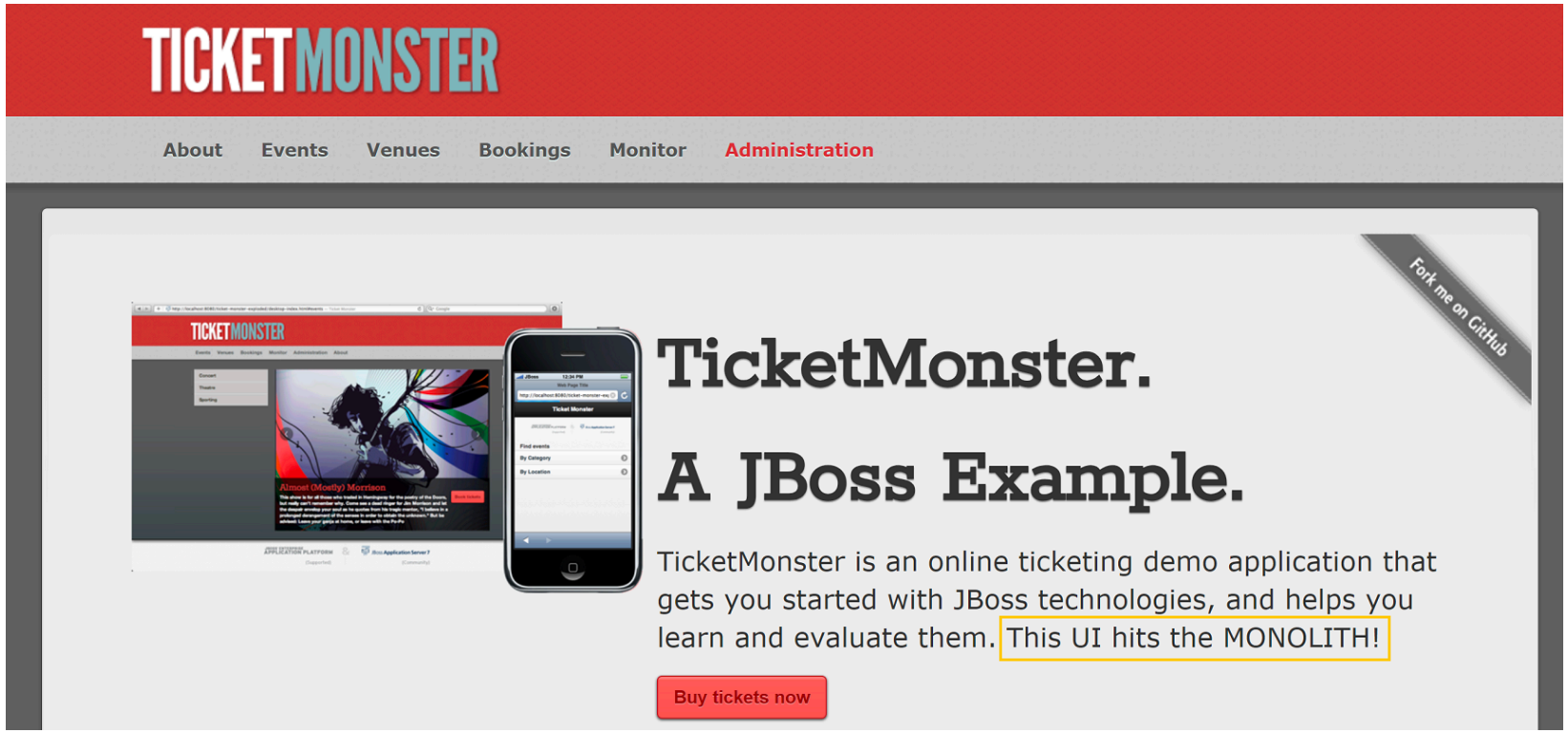
Lab 2: Extract the UI from the Monolith



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”



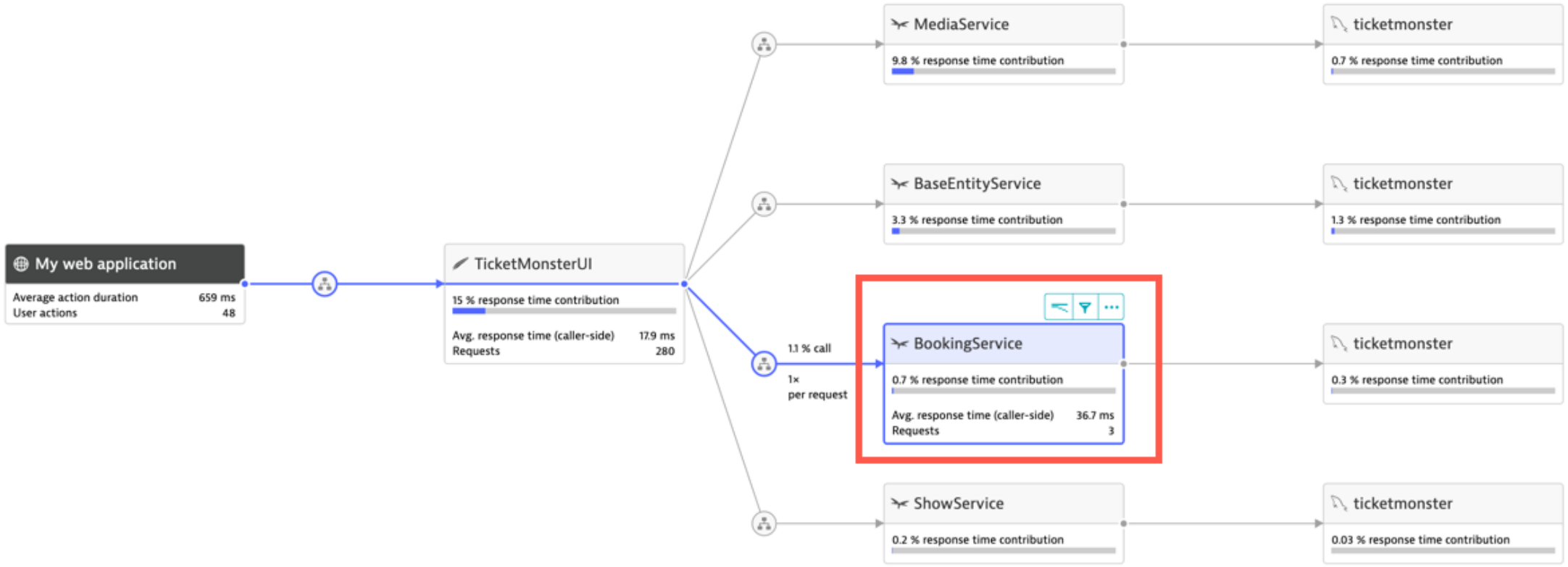
https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/2_Extract_UI_From_Monolith

How to identify a Microservice?

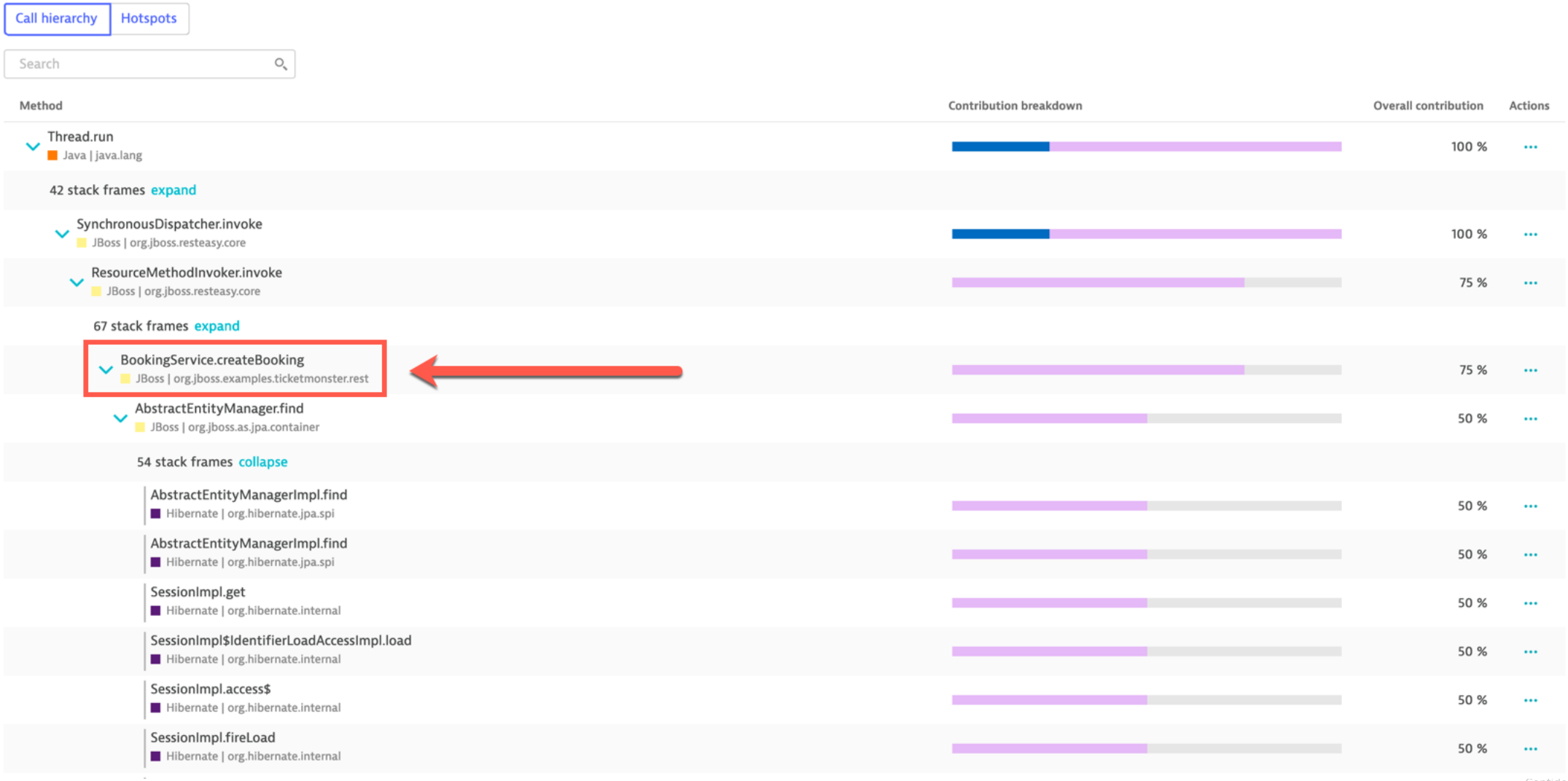
- Problem: we don't know much about our monolith:
  - Who is depending on us and how are they depending on us?
  - Whom are we depending on and how are we depending on them?
  - What happens within our monolith code base when it gets called?

- 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

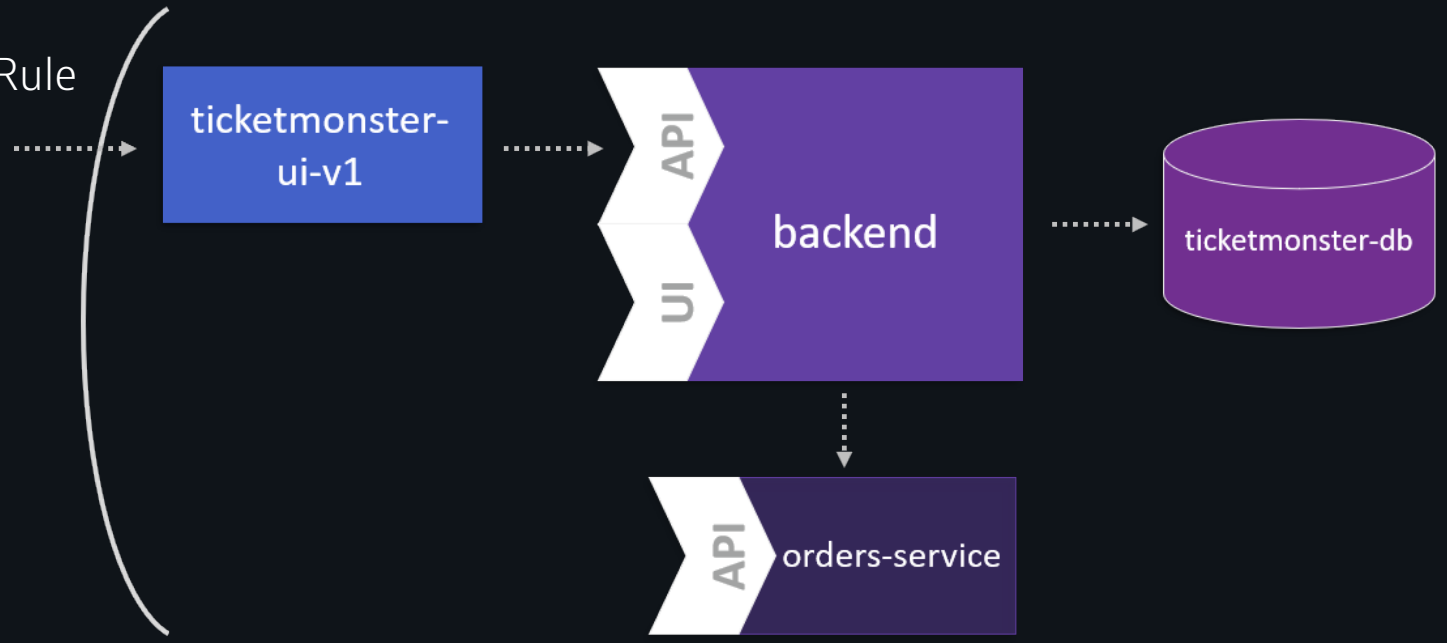


Other considerations to factor in the decision

- 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.

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/4_Identify_a_Microservice

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"

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/4_Identify_a_Microservice

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

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/4_Identify_a_Microservice

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

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/4_Identify_a_Microservice

Step 5:

Consider Service Flow in Dynatrace

a. Log into Dynatrace console and select the Project-## Management Zone

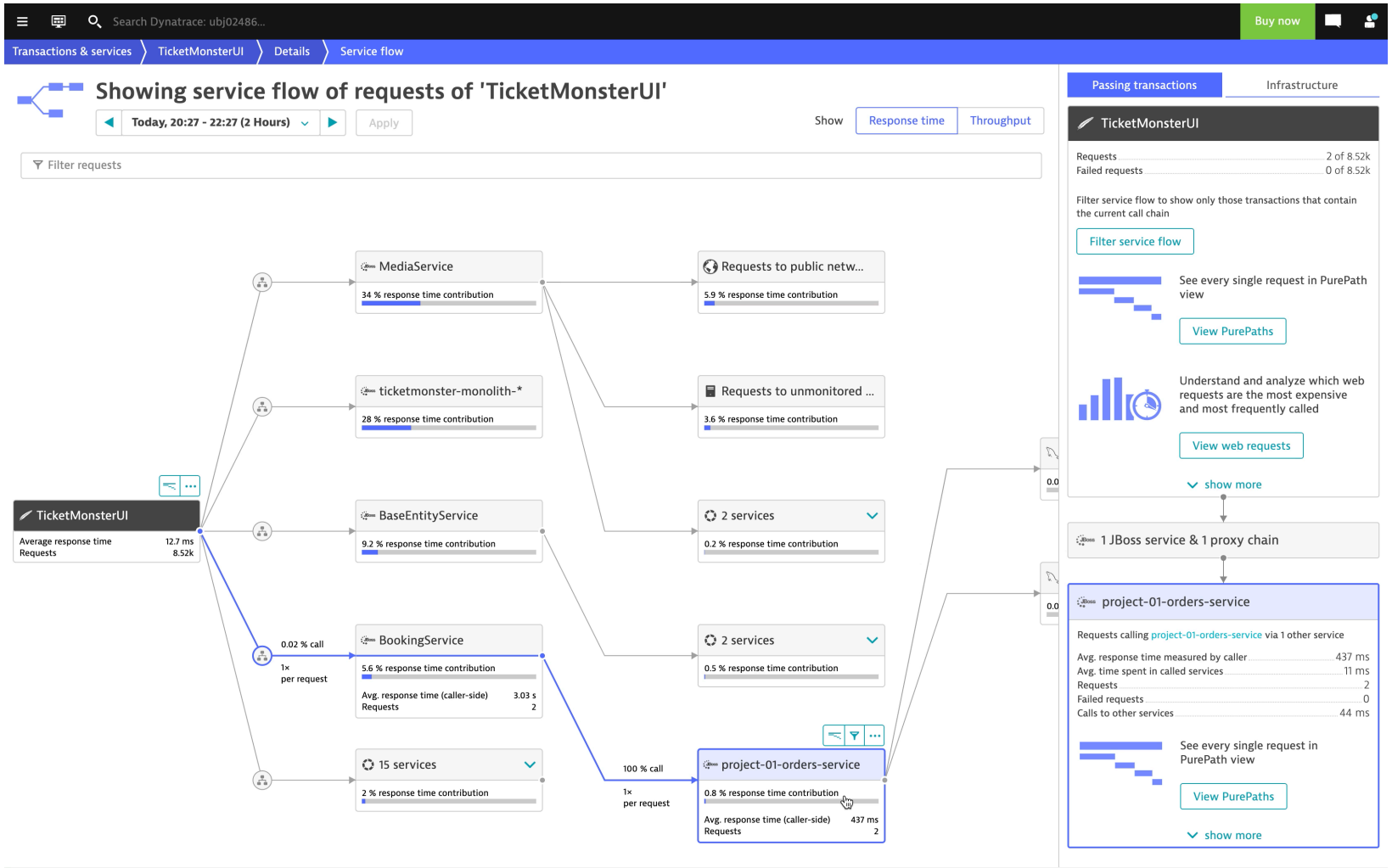
b. Choose the Transactions & services tab from left menu

c. Select TicketMonsterUI and click on View service flow

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/4_Identify_a_Microservice

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

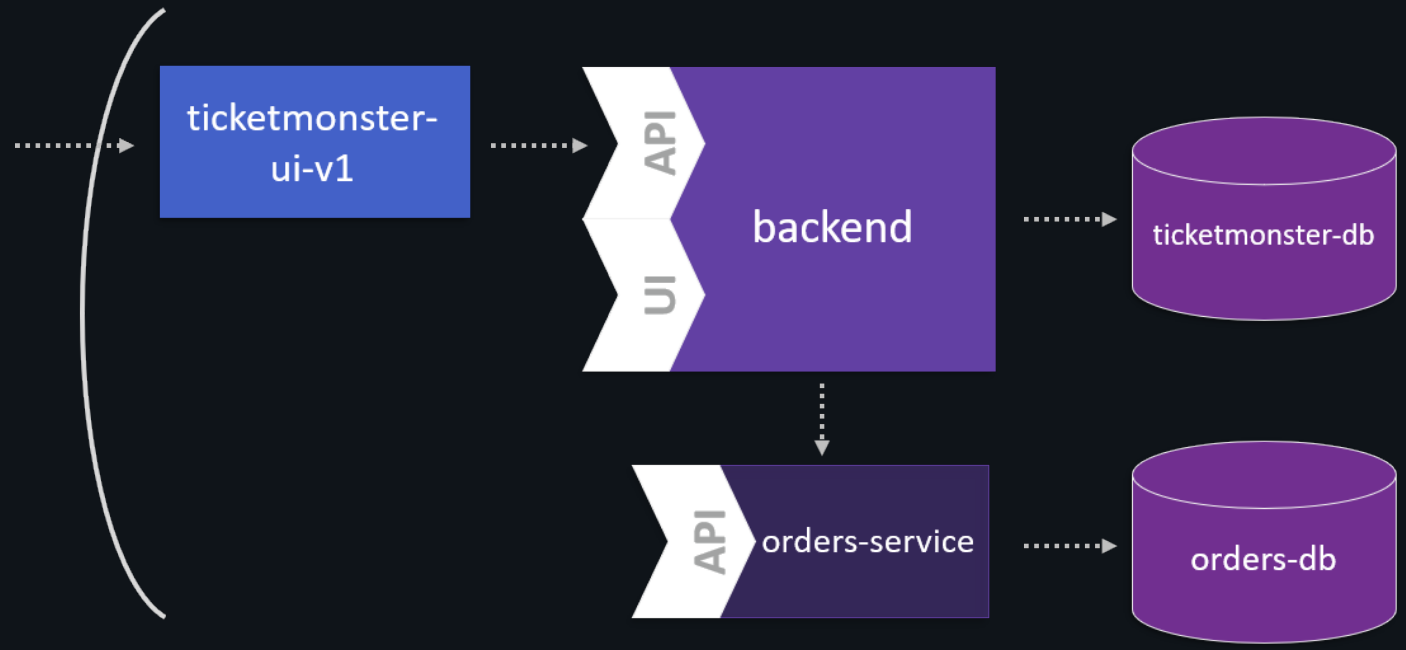
- Data model shows how data is stored and entities relate to each other in the persistence layer.
- Domain model describes the *behavior* of the solution space of a microservice's domain and tends to focus on use case scenarios

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”

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/4_Identify_a_Microservice

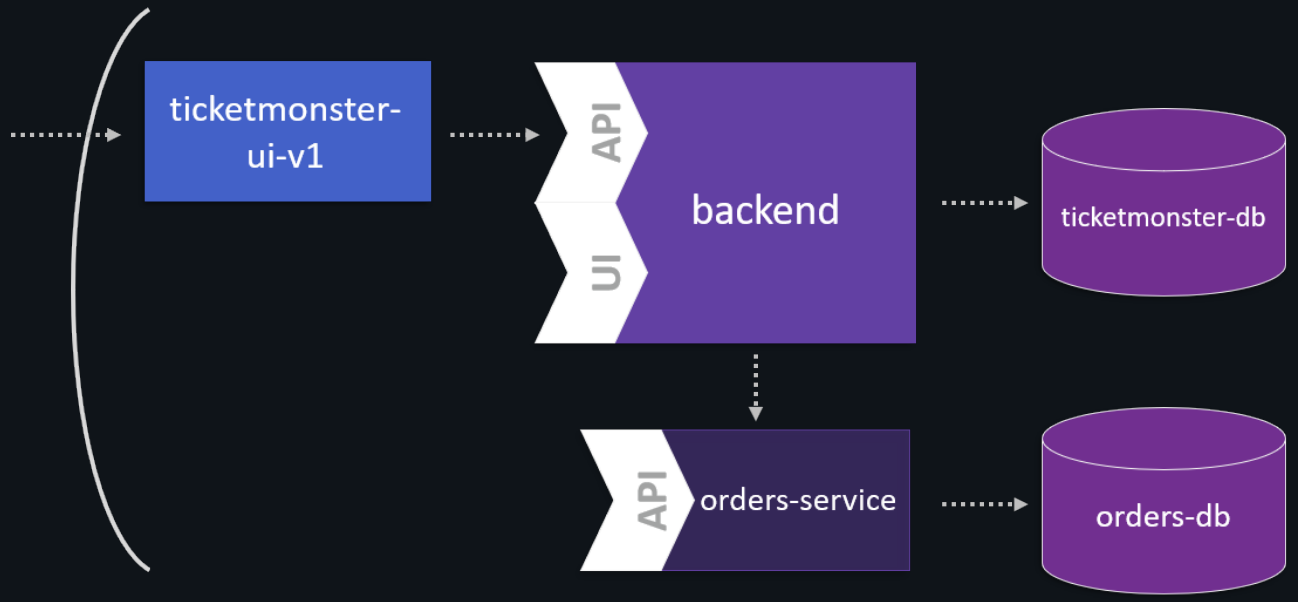
Step 2: View database statements

Statement	Executions	Total executions	Actions
select section0_.id as id1_6_0_, section0_.description as descript2_6_0_, section0_.name as name3_6_0_, section0_.numberOfRows as numberOf4_6_0_, section0_.rowCapacity as rowCapac5_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 mediaIte8_11_1_, venue1_.name as name7_11_1_, mediaitem2_.id as id1_4_2_, mediaitem2_.mediaType as	0.03 /min	41	
update ticket	0.01 /min	16	
insert into ticket	0.01 /min	16	
select ticketcate0_.id as id1_9_0_, ticketcate0_.description as descript2_9_0_ from TicketCategory ticketcate0_ where ticketcate0_.id=?	0.01 /min	11	
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 mediaIte5_2_2_, event2_.name as name3_2_2_, ticketpric3_.show_id as show_id4_10_3_, ticketpric3_.id as id1_10_3_, ticketpric3_.id as id1_10_4_, ticketpric3_.price as price2_10_4_, ticketpric3_.section_id as section_3_10_4_, ticketpric3_.show_id	0.01 /min	8	
select eventcateg0_.id as id1_3_0_, eventcateg0_.description as descript2_3_0_ from EventCategory eventcateg0_ where eventcateg0_.id=?	0.01 /min	8	
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 numberOf4_6_1_, sections0_.rowCapacity as rowCapac5_6_1_, sections0_.venue_id as venue_id6_6_1_ from Section sections0_ where sections0_.venue_id=?	0.01 /min	8	
select mediaitem0_.id as id1_4_0_, mediaitem0_.mediaType as mediaTyp2_4_0_, mediaitem0_.url as url3_4_0_ from MediaItem mediaitem0_ where mediaitem0_.id=?	0.01 /min	8	
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	0.01 /min	8	
select sectionall0_.id as id1_7_, sectionall0_.allocated as allocate2_7_, sectionall0_.occupiedCount as occupied3_7_, sectionall0_.performance_id as performa5_7_, sectionall0_.section_id as section_6_7_, sectionall0_.version as version4_7_ from SectionAllocation sectionall0_ where sectionall0_.performance_id=? and sectionall0_.section_id=?	0 /min	6	
select id from SectionAllocation where id =? and version =? for update	0 /min	6	
update sectionallocation	0 /min	6	
insert into booking	0 /min	6	
sql commit	0 /min	6	
select ticketpric0_.id as id1_10_, ticketpric0_.price as price2_10_, ticketpric0_.section_id as section_3_10_, ticketpric0_.show_id as show_id4_10_, ticketpric0_.ticketCategory_id as ticketCa5_10_ from TicketPrice ticketpric0_ where ticketpric0_.id in (?)	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

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/5_Domain_Model_of_Microservice

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

https://github.com/steve-caron-dynatrace/Monolith_to_Microservices/tree/master/5_Domain_Model_of_Microservice

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

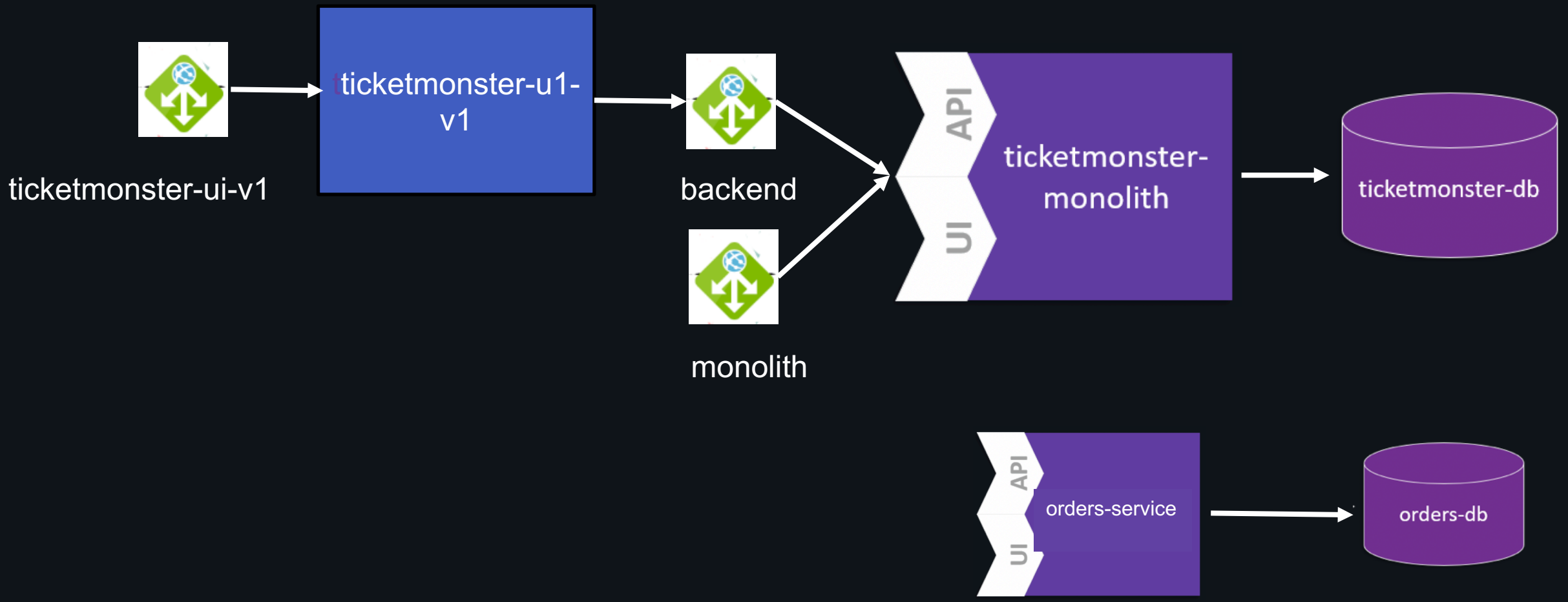
c. Copy RAW YAML

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**

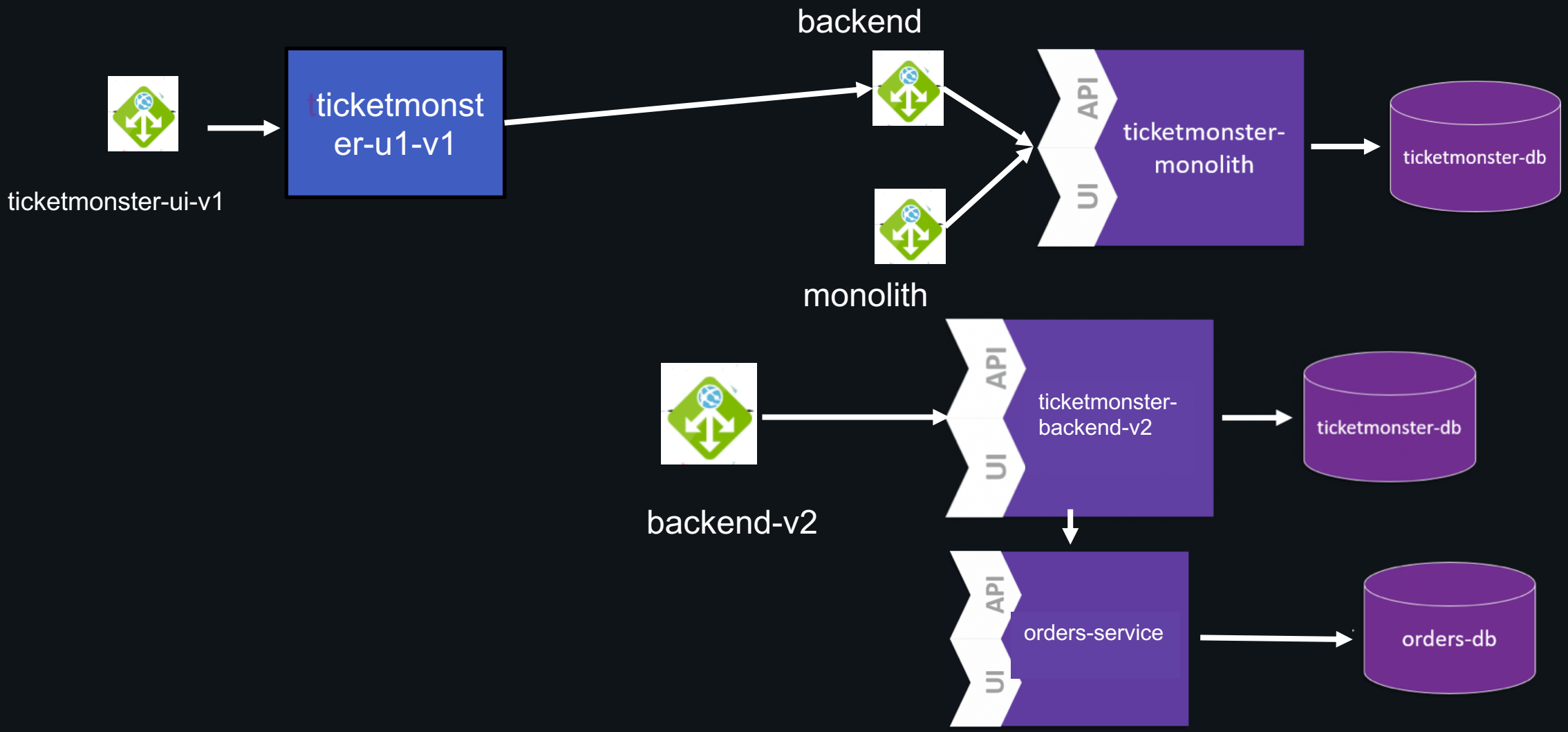
d. **Service: ticketmonster-backend-v2**

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

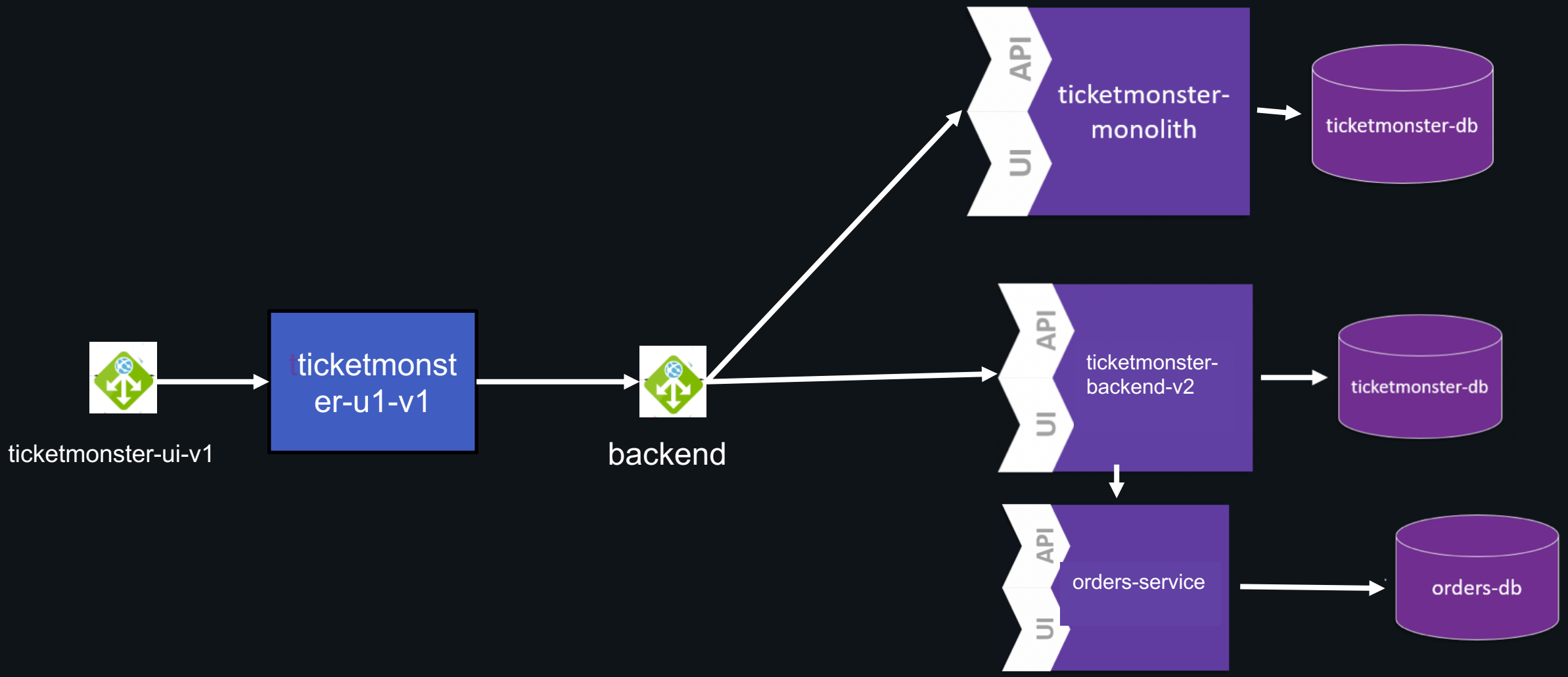
Or can also be done by editing the yaml in the console:

- 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.

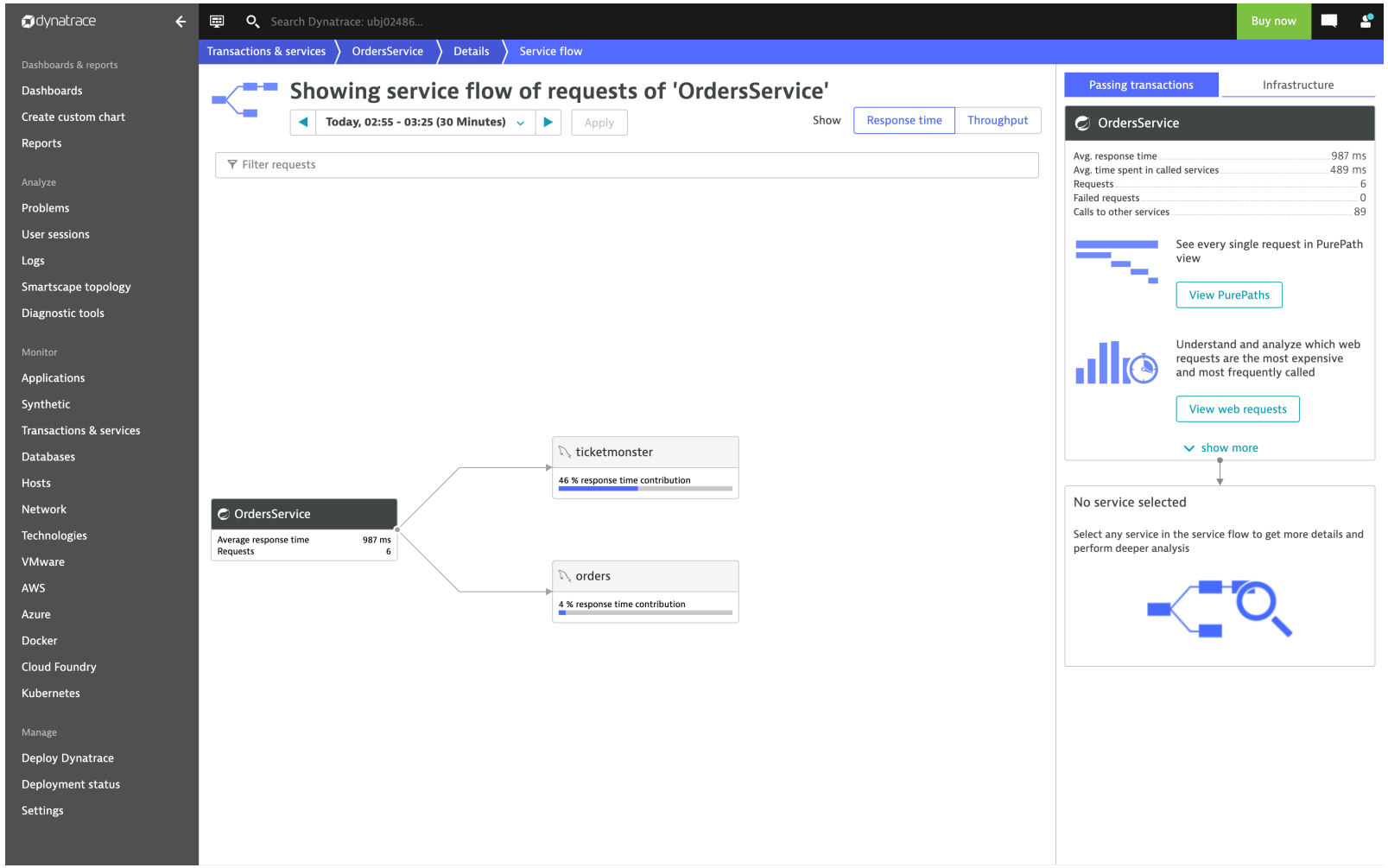
Click on View service flow.

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



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