

Spring Boot Instrumentation with the Elastic APM Java Agent

Introduction	2
Start the application for this lab	3
Auto-instrumentation of APM Java agent	5
Instrumentation with @CaptureSpan	11
Capturing errors in your instrumentation	18
Summary	21

Introduction

Welcome to the Elastic APM Java Agent lab! Along with other APM agents for Node.js, Python, Ruby, JavaScript/RUM, and Go, the Elastic APM Java Agent is part of an application performance monitoring (APM) solution from Elastic that helps you to gain insights into the performance of your Java-based application and track errors.

The Elastic APM Java Agent automatically instruments various APIs, frameworks, and application servers, such as Spring Boot 1.5.x-2.x, as well as various application servers and servlet containers like WildFly and Tomcat. Please refer to our official documentation page for an extensive list of supported technology:

<https://www.elastic.co/guide/en/apm/agent/java/current/supported-technologies-details.html>

If your favorite technology is not supported for auto-instrumentation yet, we would like to invite you to participate in our survey or contribute to the Elastic APM community:

- Survey form:
https://docs.google.com/forms/d/e/1FAIpQLScd0RYiwZGrEuxykYkv9z8HI3exx_LKCtjsqEo1OWx8BkLrOQ/viewform
- Contribution guide:
<https://github.com/elastic/apm-agent-java/blob/master/CONTRIBUTING.md>

Another option is to add the Java agent's public API as a dependency and programmatically create custom transactions and spans to instrument your application code for performance metrics and runtime errors.

In this lab, you will gain hands-on experience for both approaches. You will first set up appropriate source code branches and start the sample Spring Boot application for the lab. Next, you will configure the Java agent for auto-instrumentation and see what the APM data from your application looks like via the APM UI in Kibana. Then, you will work on extending the out-of-the-box instrumentation with your own spans collecting more fine-grained contextual information from the sample application. Finally, you will learn how to capture runtime errors with the Elastic APM Java Agent.

Now let's get started!

Start the application for this lab

Remember when you went through Lab 1, the entire sample application was fully instrumented to showcase the distributed tracing capability of Elastic APM? To go through this lab, we first need to remove all existing APM instrumentations in place in the Spring Boot component in order to start from a clean state without any Java APM instrumentation.

You can achieve that simply by checking out the code branch **java**, which is the version of the sample application that has all other components (Node.js, RUM, Python, etc.) instrumented with Elastic APM agents except the Spring Boot application managing owners, pets, ownership, and clinic visits.

Use **Ctrl+C** to stop the running Spring Boot component of the sample application from the terminal window where you started it previously in Lab 1. Then, checkout the **java** branch.

```
$ git checkout java
Switched to branch 'java'

$ git status
On branch java
nothing to commit, working tree clean
```

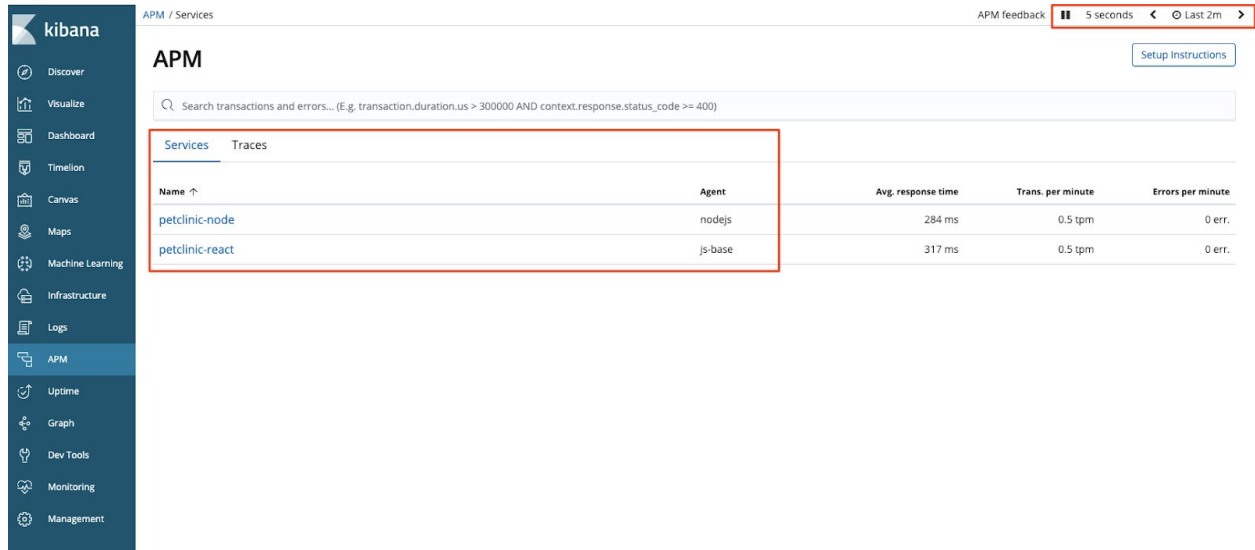
Make sure you are in the root directory of the sample application (i.e., the root directory of the cloned Github repository in your Strigo lab host). Run the following command to start the Spring Boot component again.

```
$ ./mvnw spring-boot:run
[INFO] Scanning for projects...
<...abbreviated output...>
```

Go to **http://LAB_HOST:8081** in a browser and verify the application is working. You should see the welcome page of the petclinic application. Feel free to click around.

Now, go to the APM UI inside Kibana. Update the time picker to the last 2 minutes. Notice that there is no **spring-petclinic** service presented as part of the application stack. This is because there is no APM instrumentation in the **java** code branch. You might see other services shown up in the APM UI when clicking around the petclinic web application, such as

petclinic-node and **petclinic-react**, since those components remain unchanged and fully instrumented.



The screenshot shows the Kibana APM Services page. The left sidebar contains the Kibana navigation menu with 'APM' selected. The main content area is titled 'APM' and shows a table of services. The table has columns for Name, Agent, Avg. response time, Trans. per minute, and Errors per minute. Two services are listed: petclinic-node (nodejs) and petclinic-react (js-base). The table is highlighted with a red border.

Name ↑	Agent	Avg. response time	Trans. per minute	Errors per minute
petclinic-node	nodejs	284 ms	0.5 tpm	0 err.
petclinic-react	js-base	317 ms	0.5 tpm	0 err.

Next up, we will configure the Elastic APM Java Agent to auto-instrument the Spring Boot component.

Auto-instrumentation of APM Java agent

Setting up the Elastic APM Java Agent is as simple as adding the `-javaagent` flag and a couple of system properties to your JVM startup command. As long as you are using supported frameworks and platforms, such as Spring MVC/Spring Boot, Servlet API, and JAX-RS, your application will be instrumented and monitored automatically.

Use **Ctrl+C** to stop the running Spring Boot component. Open and edit the `mvnw` file under the root of the cloned Github repository directory to the following. This is towards the end of the file.

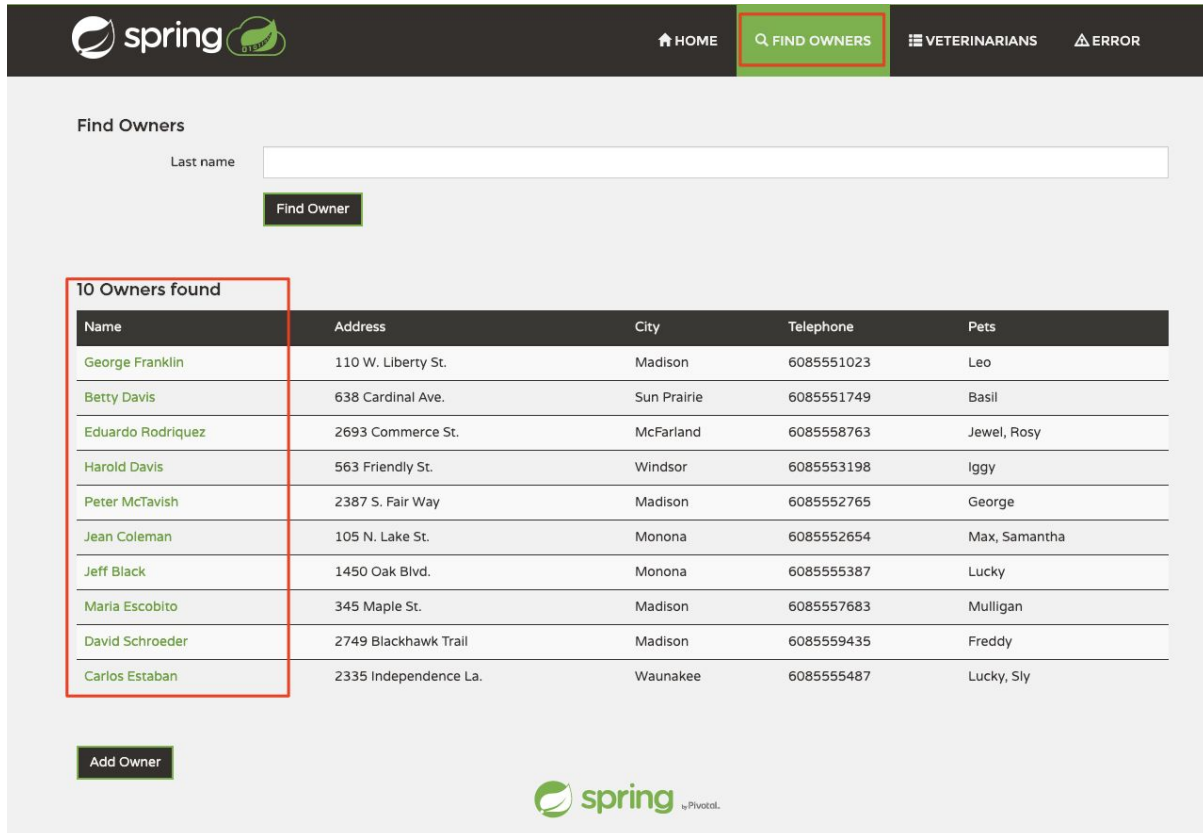
```
<...abbreviated contents from mvnw file...>

exec "$JAVACMD" \
  -javaagent:./agents/elastic-apm-agent-1.6.0.jar \
  -Delastic.apm.application_packages=org.springframework.samples.petclinic \
  -Delastic.apm.service_name=spring-petclinic \
  -Delastic.apm.service_version=1.0.0 \
  -Delastic.apm.server_urls=$ELASTIC_APM_SERVER_URL \
  -Delastic.apm.secret_token=$ELASTIC_APM_SECRET_TOKEN \
  -Delastic.apm.verify_server_cert=false \
  -Delastic.apm.ignore_urls=/health,/metrics*,/jolokia \
  $MAVEN_OPTS \
  -classpath "$MAVEN_PROJECTBASEDIR/.mvn/wrapper/maven-wrapper.jar" \
  "-Dmaven.home=${M2_HOME}"
"-Dmaven.multiModuleProjectDirectory=${MAVEN_PROJECTBASEDIR}" \
  ${WRAPPER_LAUNCHER} $MAVEN_CMD_LINE_ARGS
```

Make sure you are in the root directory of the sample application (i.e., the root directory of the cloned Github repository in your Strigo lab host). Run the following command to start the Spring Boot component again.

```
$ ./mvnw spring-boot:run
[INFO] Scanning for projects...
<...abbreviated output...>
```

Go to `http://LAB_HOST:8081` in a browser and verify the application is working. You should see the welcome page of the petclinic application. Navigate to the **FIND OWNERS** page in the application web UI, where you should see a list of owners.



Find Owners

Last name

Find Owner

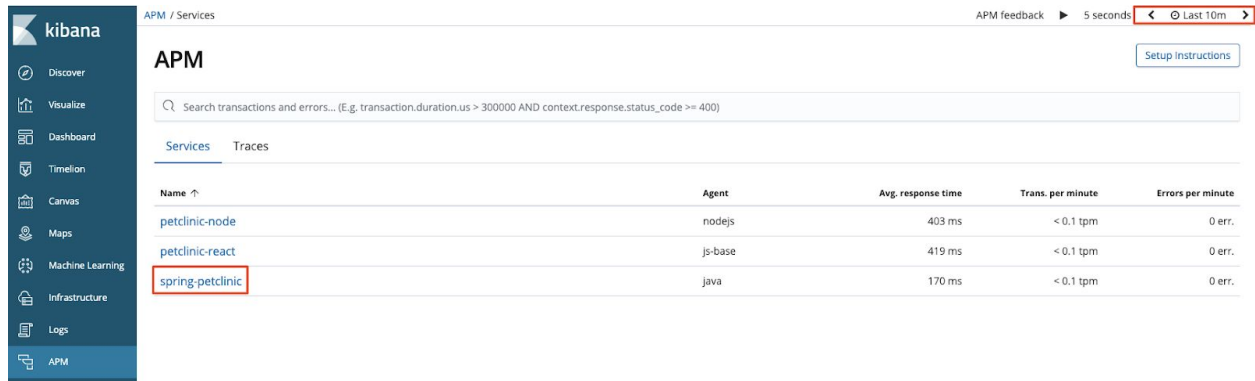
10 Owners found

Name	Address	City	Telephone	Pets
George Franklin	110 W. Liberty St.	Madison	6085551023	Leo
Betty Davis	638 Cardinal Ave.	Sun Prairie	6085551749	Basil
Eduardo Rodriguez	2693 Commerce St.	McFarland	6085558763	Jewel, Rosy
Harold Davis	563 Friendly St.	Windsor	6085553198	Iggy
Peter McTavish	2387 S. Fair Way	Madison	6085552765	George
Jean Coleman	105 N. Lake St.	Monona	6085552654	Max, Samantha
Jeff Black	1450 Oak Blvd.	Monona	608555387	Lucky
Maria Escobito	345 Maple St.	Madison	6085557683	Mulligan
David Schroeder	2749 Blackhawk Trail	Madison	6085559435	Freddy
Carlos Estaban	2335 Independence La.	Waunakee	6085555487	Lucky, Sly

Add Owner

spring by Pivotal

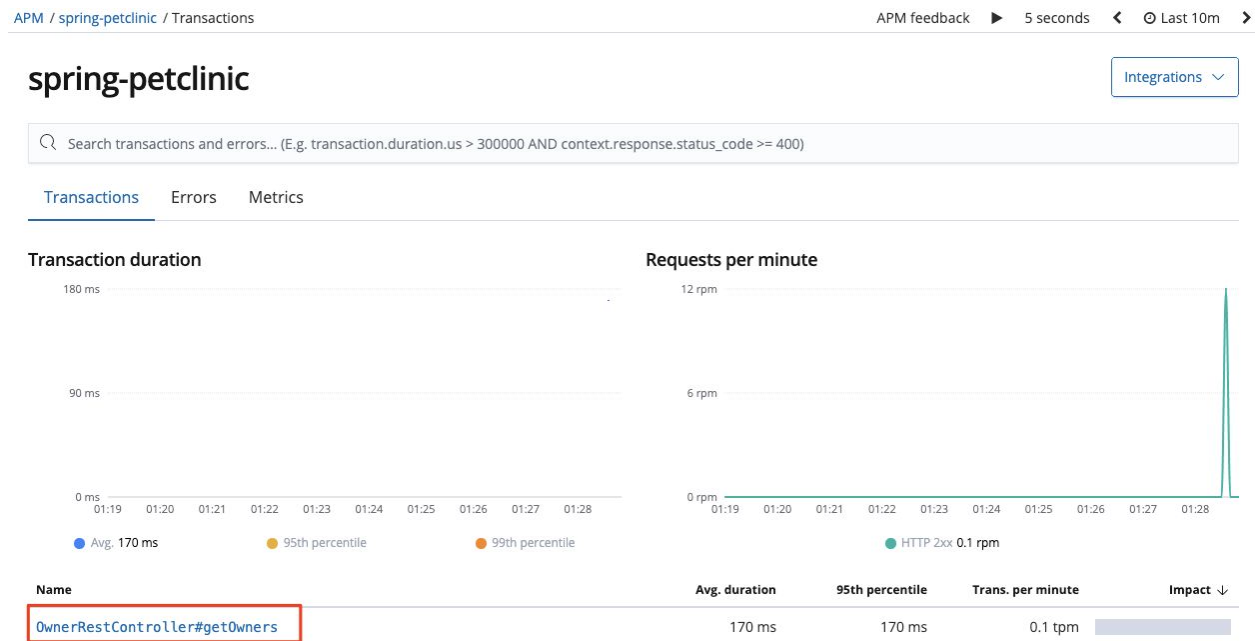
Now, navigate to the Kibana API UI and you should be able to see the **spring-petclinic** service as part of the application stack. Make sure you have the time picker set properly.



The screenshot shows the Kibana APM interface. On the left is a sidebar with navigation links: Discover, Visualize, Dashboard, Timeline, Canvas, Maps, Machine Learning, Infrastructure, Logs, and APM. The main header shows 'APM / Services'. Below the header is a search bar with the text 'Search transactions and errors... (E.g. transaction.duration.us > 300000 AND context.response.status_code >= 400)'. Below the search bar is a table of services:

Name	Agent	Avg. response time	Trans. per minute	Errors per minute
petclinic-node	nodejs	403 ms	< 0.1 tpm	0 err.
petclinic-react	js-base	419 ms	< 0.1 tpm	0 err.
spring-petclinic	java	170 ms	< 0.1 tpm	0 err.

Clicking on the **spring-petclinic** service will take you to the Transactions page, where all transactions captured out of the box by the Java APM agent will be listed. Look for the transaction named **OwnerRestController#getOwners**. By default, transactions and spans from the Java APM agent will have a name in the format of **ClassName#methodName**. So, the fact that this transaction shows up in the APM UI immediately tells us the actual code path being executed in the Spring Boot component when clicking on the petclinic application web UI.

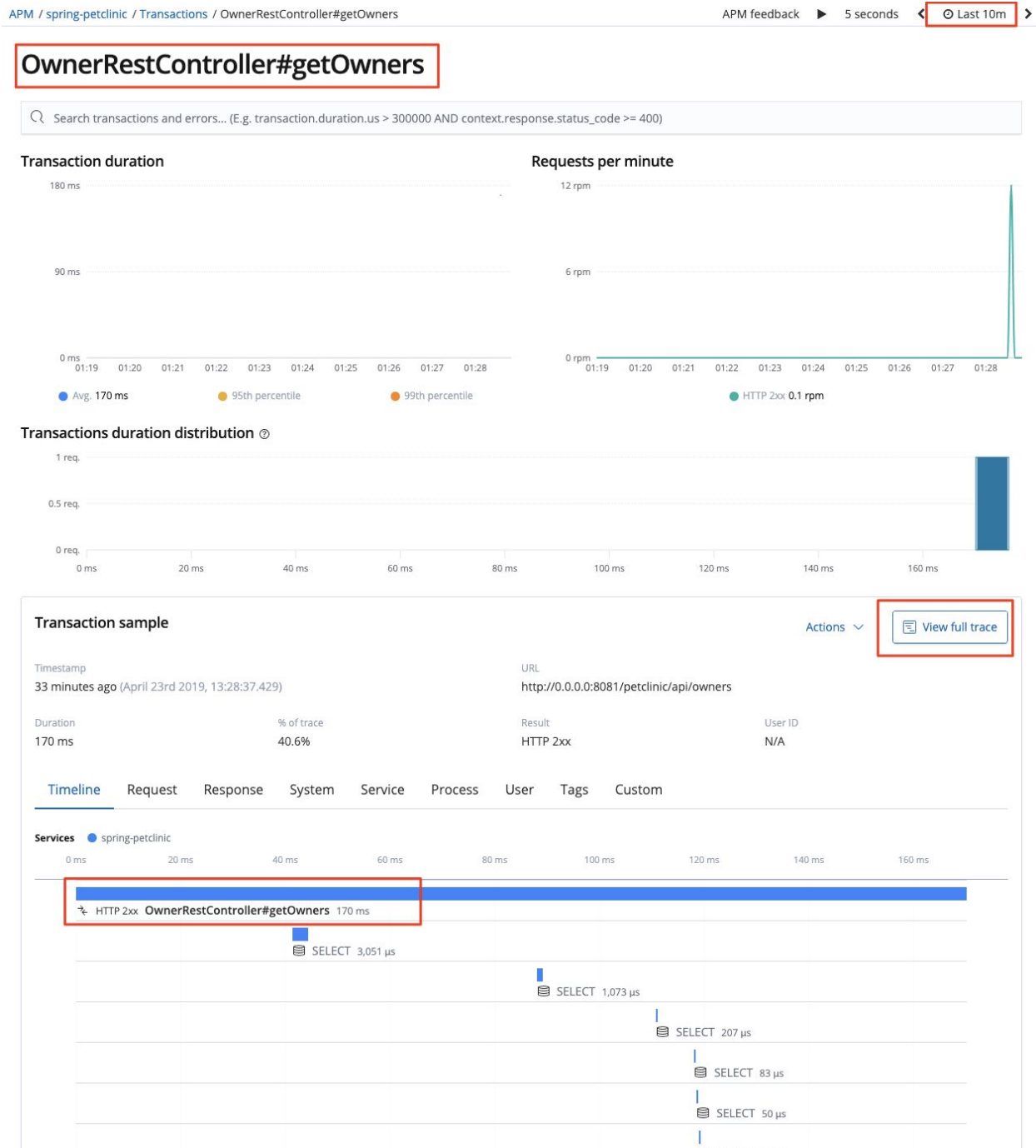


The screenshot shows the Kibana APM Transactions page for the 'spring-petclinic' service. The header shows 'APM / spring-petclinic / Transactions'. Below the header is a search bar with the text 'Search transactions and errors... (E.g. transaction.duration.us > 300000 AND context.response.status_code >= 400)'. Below the search bar is a table of transactions:

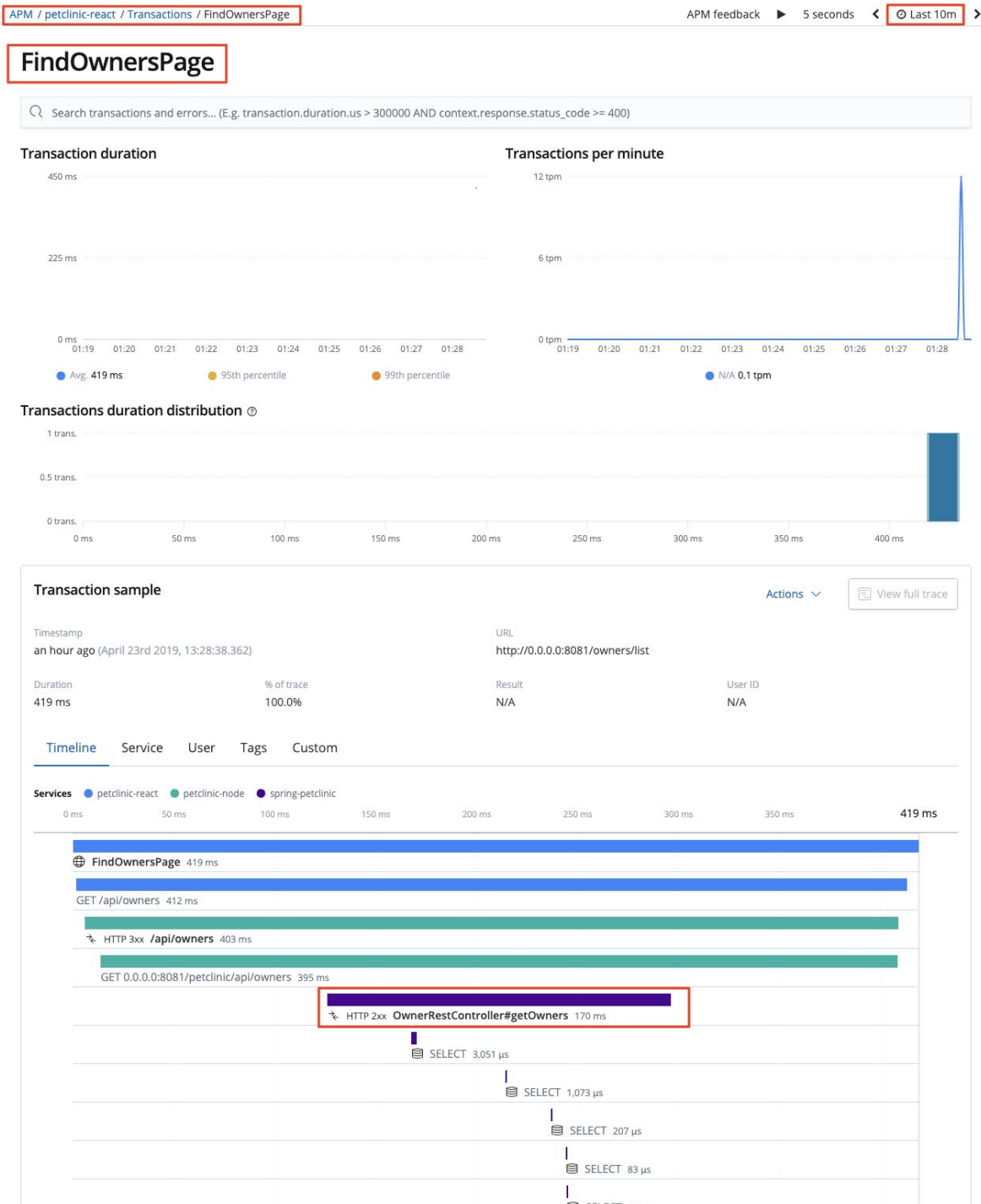
Name	Avg. duration	95th percentile	Trans. per minute	Impact
OwnerRestController#getOwners	170 ms	170 ms	0.1 tpm	

You can see more information about the **OwnerRestController#getOwners** transaction by clicking on it and going to the details page. If you have finished Lab 1, most data presented

on this page should look familiar to you. One thing to note is the **View full trace** button in the **Transaction Sample** section.



It will lead you to the origin of the distributed trace of this microservice-based sample application where this particular Spring Boot transaction sits. In this case, everything starts from the **FindOwnersPage** in the **petclinic-react** component, then hops onto the Node.js API proxy layer, and eventually hits the **getOwners** method of the **OwnerRestController** class in the Spring Boot component.



Now that we see how easy it is to set up the Java APM agent for auto-instrumentation, let's move onto the next section and see how we can extend it with customizations.

Instrumentation with @CaptureSpan

In this section, we will use the `@CaptureSpan` annotation API (<https://www.elastic.co/guide/en/apm/agent/java/current/public-api.html#api-capture-span>) to create custom spans in the Spring Boot component.

First, let's stop the application with **Ctrl+C** in the terminal window. Then open the file `src/main/java/org/springframework/samples/petclinic/rest/OwnerRestController.java`, and import the required Elastic APM class.

```
import co.elastic.apm.api.CaptureSpan;
```

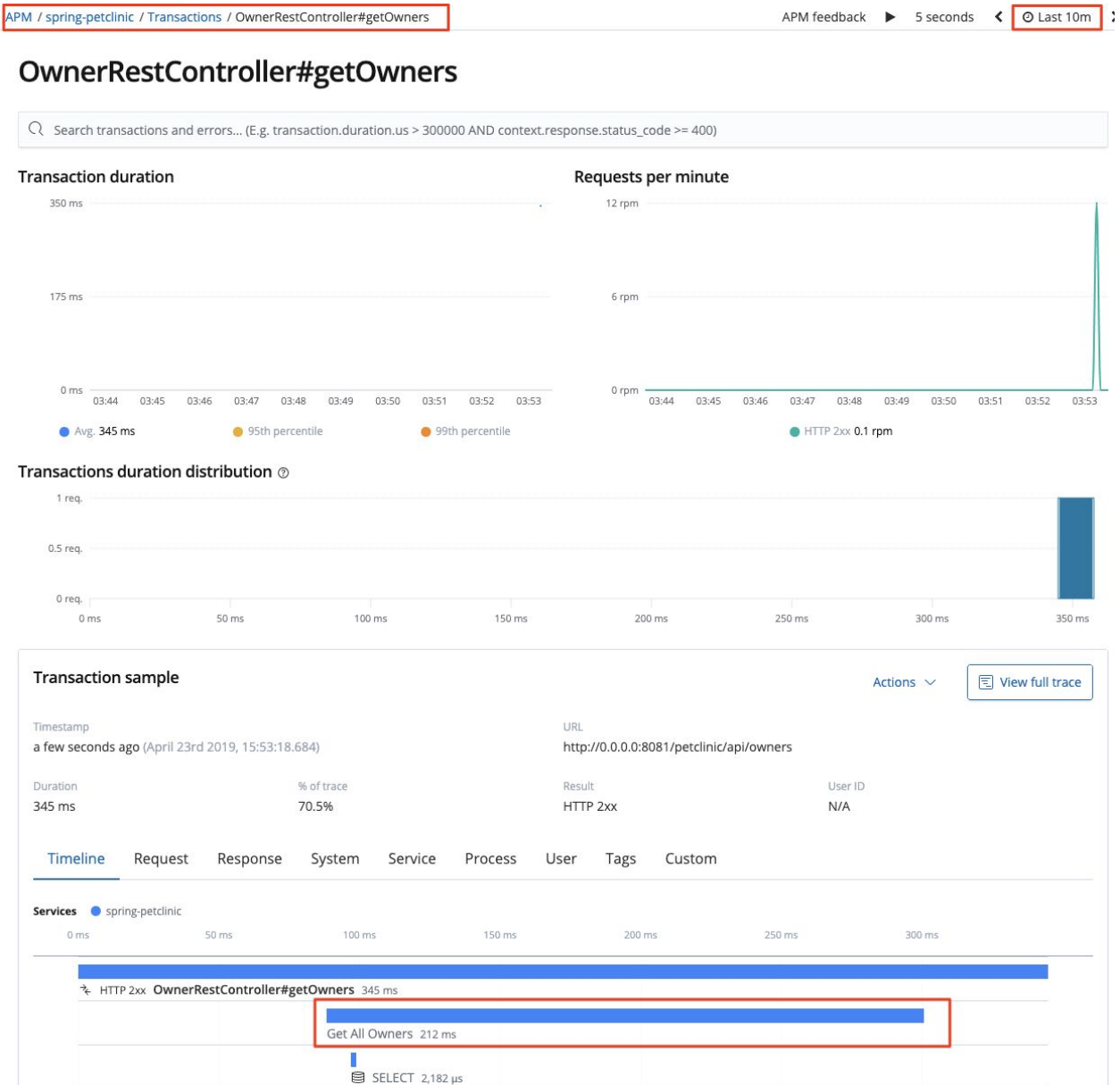
From the previous section we know that visiting the **FIND OWNERS** page will execute the `getOwners(int ownerId)` method in this controller class. Annotate it with `@CaptureSpan` as follows (nothing in the body of this method changes):

```
@CaptureSpan(value = "Get All Owners")
@PreAuthorize( "hasRole(@roles.OWNER_ADMIN)" )
@RequestMapping(value = "", method = RequestMethod.GET, produces =
MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<Collection<Owner>> getOwners() {
    Collection<Owner> owners = this.clinicService.findAllOwners();
    if (owners.isEmpty()) {
        return new
ResponseEntity<Collection<Owner>>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<Collection<Owner>>(owners,
HttpStatus.OK);
}
```

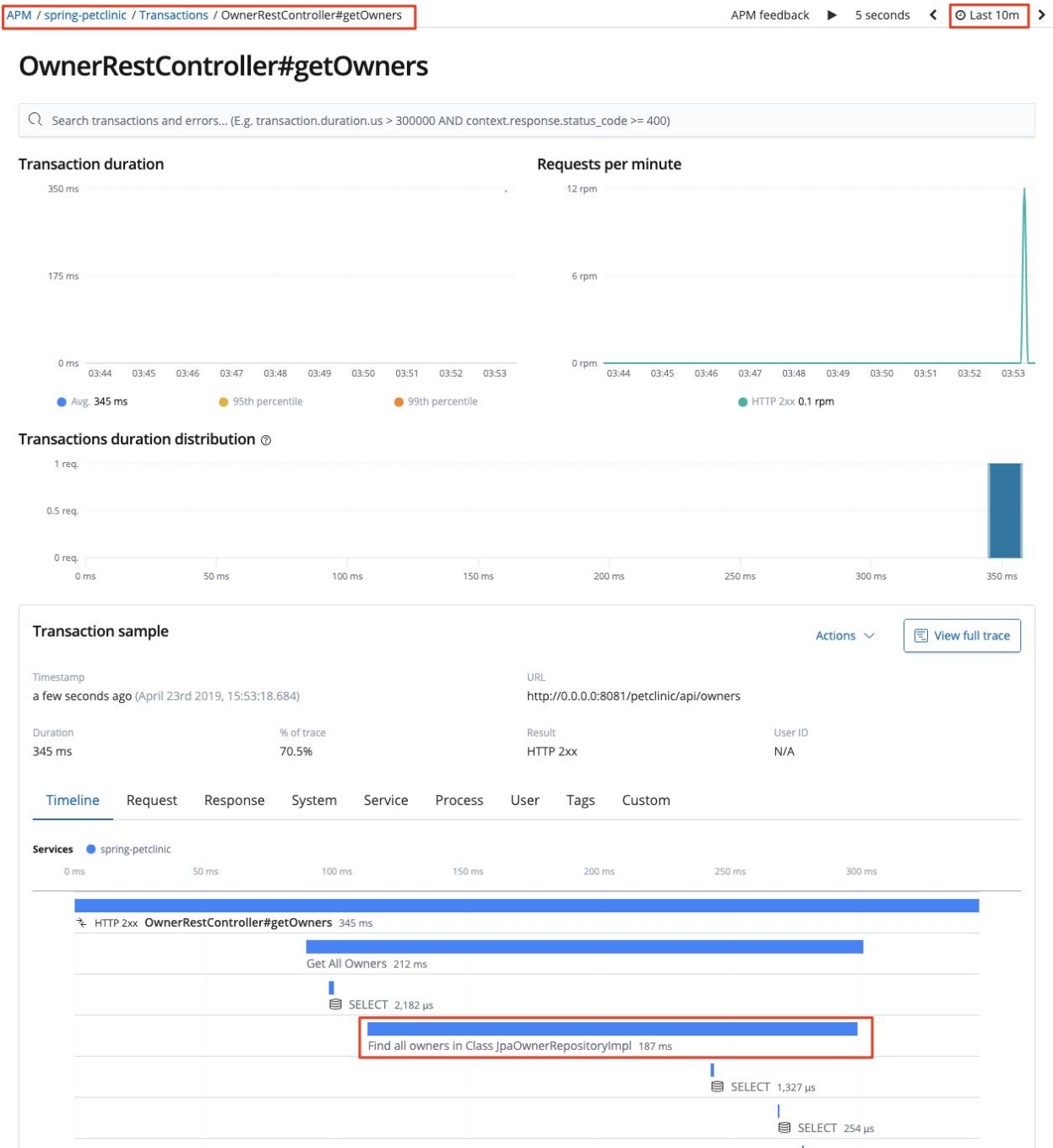
Save the file and make sure you are in the root directory of the sample application (i.e., the root directory of the cloned Github repository in your Strigo lab host). Run the following command to start the Spring Boot component again.

```
$ ./mvnw spring-boot:run
[INFO] Scanning for projects...
<...abbreviated output...>
```

Go to **http://LAB_HOST:8081** in a browser and verify the application is working. You should see the welcome page of the petclinic application. Navigate to the **FIND OWNERS** page in the application web UI again. Switch to the details page of the **OwnerRestController#getOwners** transaction, you will now see the custom span created from the **@CaptureSpan** annotation shows up in the wallfall view.



You just instrumented a method inside of a REST controller class, with just a simple Java annotation! With `@CaptureSpan`, you can actually instrument other places along the code path in the Spring Boot component. Here is a take-home challenge for you: try to create another custom span with `@CaptureSpan` in the Spring Repository layer with end results like the following:



(Hint: the solution is in the **master** branch of the cloned Github repository.)

So much fun with `@CaptureSpan`, even in its simplest form! However, sometimes it is useful to capture some additional contextual data with the current span. For this we can use tags. For example, you might want to grab certain aspects of the pet owner and set them as tags for troubleshooting and/or debugging purposes.

See the documentation around adding tags to spans:

<https://www.elastic.co/guide/en/apm/agent/java/master/public-api.html#api-span-add-tag>

Moving next, you are going to instrument the `OwnerRestController#getOwner` method and use tags to capture information about the selected owner. Note that this method will be executed when you click on any of the owners from the **FIND OWNERS** page.

First, let's stop the application with **Ctrl+C** in the terminal window. Then open the file `src/main/java/org/springframework/samples/petclinic/rest/OwnerRestController.java`, and import the required Elastic APM classes.

```
import co.elastic.apm.api.CaptureSpan;
import co.elastic.apm.api.ElasticApm;
import co.elastic.apm.api.Span;
```

Instrument the `getOwner(int ownerId)` method as follows:

```
@CaptureSpan(value = "Find Owner By Owner ID")
@PreAuthorize("hasRole(@roles.OWNER_ADMIN)")
@RequestMapping(value = "/{ownerId}", method = RequestMethod.GET,
produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
public ResponseEntity<Owner> getOwner(@PathVariable("ownerId") int
ownerId) {
    Span span = ElasticApm.currentSpan();
    span.addLabel("OwnerID", ownerId);
    Owner owner = null;
    owner = this.clinicService.findOwnerById(ownerId);
    span.addLabel("LastName", owner.getLastName());
    span.addLabel("FirstName", owner.getFirstName());
    if (owner == null) {
        return new ResponseEntity<Owner>(HttpStatus.NOT_FOUND);
    }
}
```

```
        return new ResponseEntity<Owner>(owner, HttpStatus.OK);  
    }
```

Make sure you are in the root directory of the sample application (i.e., the root directory of the cloned Github repository in your Strigo lab host). Run the following command to start the Spring Boot component again.

```
$ ./mvnw spring-boot:run  
[INFO] Scanning for projects...  
<...abbreviated output...>
```

In the sample application web UI, navigate to the **FIND OWNERS** page and click on one of the pet owners in the table. Switch back to Kibana APM UI. You should be able to see the **OwnerRestController#getOwner** transaction with the custom span.

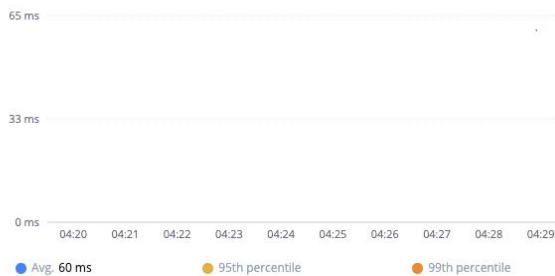
APM / spring-petclinic / Transactions / OwnerRestController#getOwner

APM feedback ▶ 5 seconds ◀ Last 10m ▶

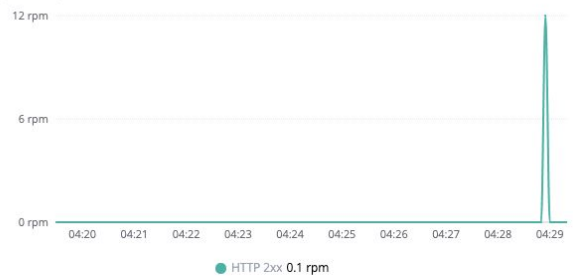
OwnerRestController#getOwner

🔍 Search transactions and errors... (E.g. transaction.duration.us > 300000 AND context.response.status_code >= 400)

Transaction duration



Requests per minute



Transactions duration distribution



Transaction sample

Actions ▾

View full trace

Timestamp

a few seconds ago (April 23rd 2019, 16:28:58.568)

URL

http://0.0.0.0:8081/petclinic/api/owners/4

Duration

60 ms

% of trace

74.1%

Result

HTTP 2xx

User ID

N/A

Timeline

Request

Response

System

Service

Process

User

Tags

Custom

Services ● spring-petclinic

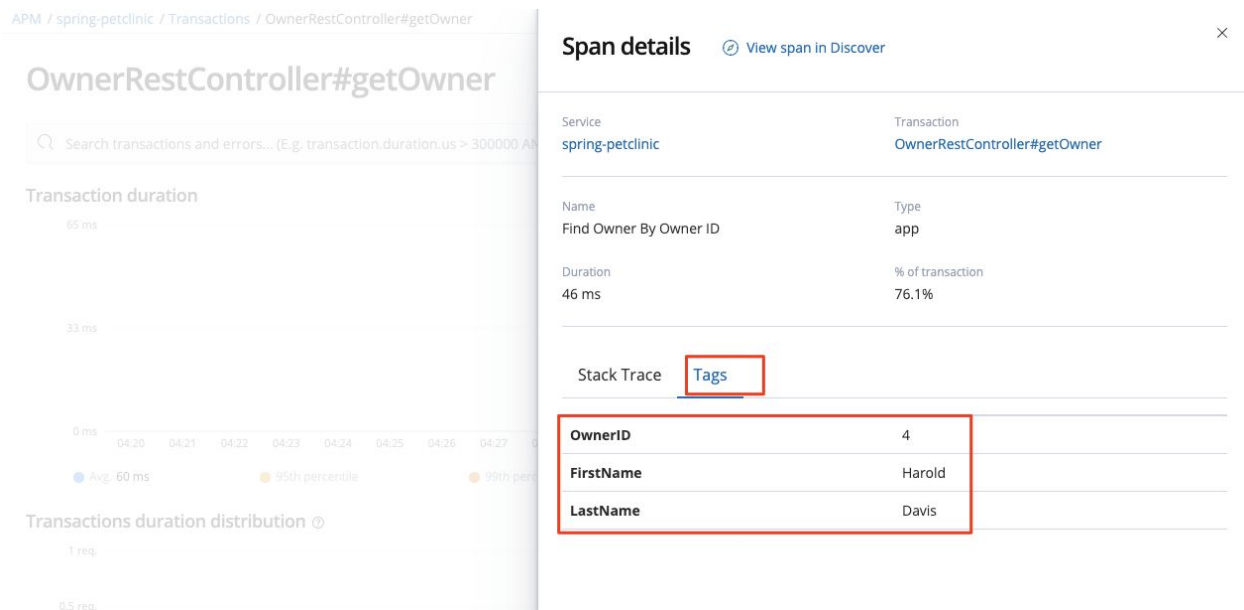
0 ms 10 ms 20 ms 30 ms 40 ms 50 ms 60 ms

HTTP 2xx OwnerRestController#getOwner 60 ms

Find Owner By Owner ID 46 ms

SELECT 390 μs

Click on the custom span to reveal the Span Details slide-out window. Then click on the Tags tab to check the owner ID passed in when invoking this method during runtime, as well as other information about the owner retrieved from the backend database.

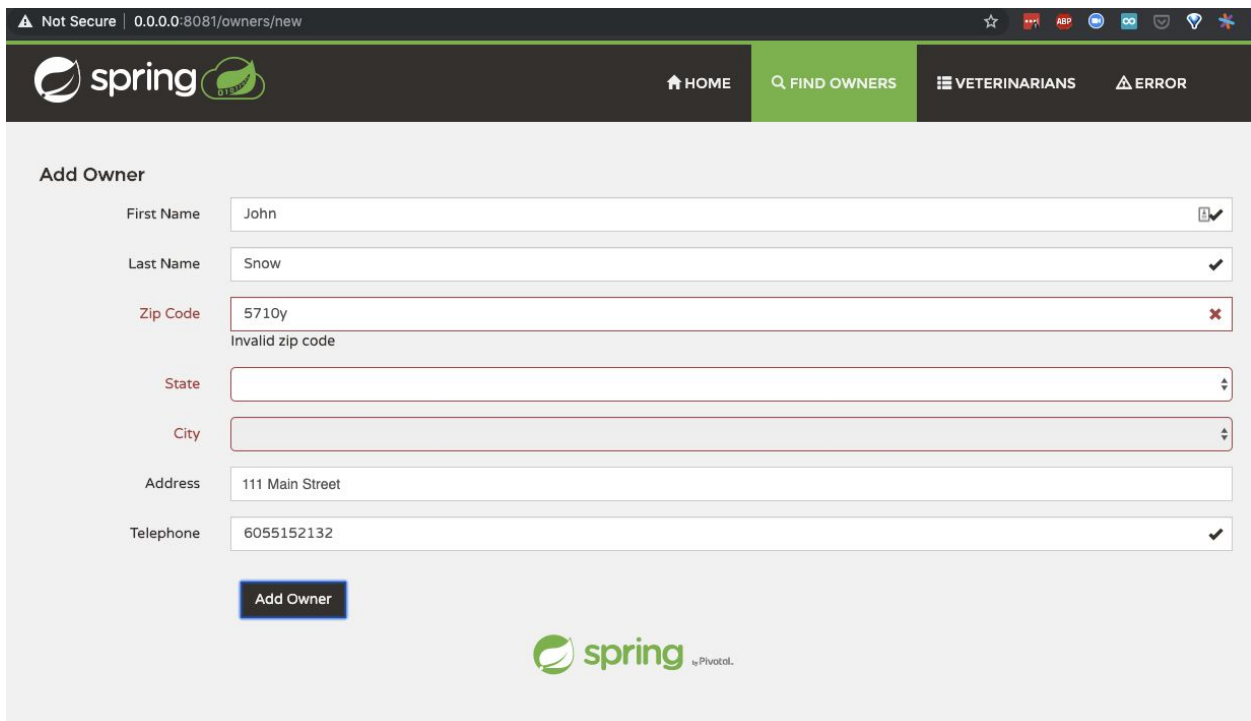


Well done! You are now a master of instrumenting with custom spans using the Java APM agent. Next, in the final section of this lab, you are going to see how the Java APM agent handles exception capturing.

Capturing errors in your instrumentation

APM Transaction objects can capture exceptions with the `captureException(Exception e)` method and report them to the APM server.

The sample petclinic application leverages Spring's validation capability (<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#validation>) for data integrity check. A specific example of validation is the zip code validity check when adding a new owner.



You are going to instrument the method where this zip code validation check happens and capture this runtime exception in the Spring Boot component using the Java APM agent.

First, let's stop the application with **Ctrl+C** in the terminal window. Then open the file `src/main/java/org/springframework/samples/petclinic/validation/zipCodeValidator.java`, and import the required Elastic APM classes.

```
import co.elastic.apm.api.CaptureSpan;
```

```
import co.elastic.apm.api.ElasticApm;
import co.elastic.apm.api.Transaction;
```

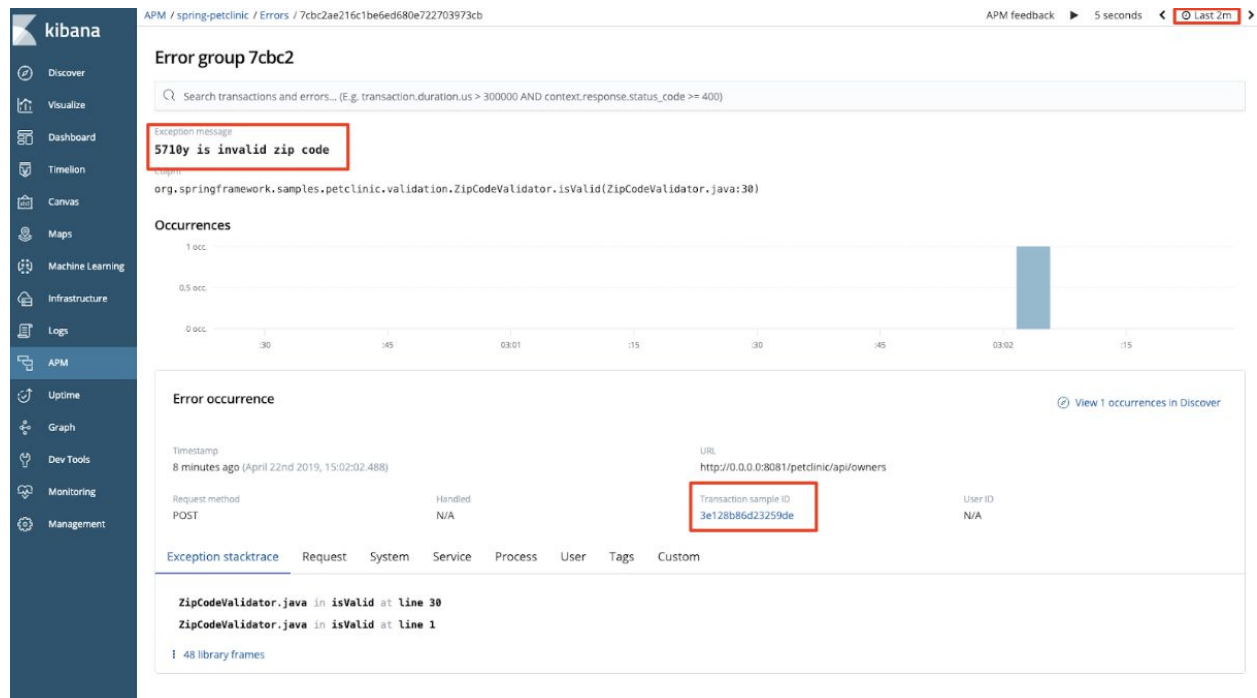
Now, instrument the `isValid(String value, ConstraintValidatorContext context)` method as follows:

```
@CaptureSpan(value = "validateZipCode")
@Override
public boolean isValid(String value, ConstraintValidatorContext
context) {
    Matcher matcher = zipPattern.matcher(value);
    boolean match = matcher.find();
    if(!match) {
        Transaction transaction = ElasticApm.currentTransaction();
        if(transaction != null) {
            transaction.captureException(new
IllegalArgumentException(String.format("%s is invalid zip code", value)));
        }
    }
    return match;
}
```

With the above instrumentation, when you click on the **Add Owner** button and put in an invalid zip code, not only will the application web UI give you warnings shown previously, but also will this error be captured and presented in the APM UI under the petclinic-spring service.



Clicking on the error will drill down to its details. Note that the ID of the transaction which captured the error is also presented.



Summary

Congratulations! You have successfully completed this lab.

To recap, the Elastic APM Java Agent automatically measures the performance of your Java-based application and tracks errors. For example, it records spans for database queries and transactions for incoming HTTP requests. By default the agent comes with support for common frameworks out of the box. To instrument other events, you can use custom spans.

At this point, you have gained hands-on experience setting up the agent with automatic out-of-the-box instrumentation, creating custom spans (even with tags), and capturing errors using the Elastic APM Java Agent. You now have a solid foundation to explore further and put the Java APM agent into your own applications.