MyDriving Reference Guide

Building Integrated IoT Systems that Collect, Process, and Visualize Data

5 May 2016

Microsoft

PUBLISHED BY
Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Contents

CHAPTER 1

Introduction

Internet-of-Things (IoT) devices are clearly setting the stage for a tremendous phase of technological innovation and expansion that will affect everyone from individual consumers to entire industries. With the number of IoT devices expected to reach into the tens of *billions* by 2020*—including PCs, tablets, and smart phones playing an IoT role—and with each one of those devices collecting and logging data, the amount of information available to us will be orders of magnitude beyond the dreams of a mere decade ago.

Data collection, however, is not the true heart of IoT. Rather, it's *effectively, efficiently, and intelligently converting that data into meaningful value for human beings in their many endeavors.* That is, the goal of IoT is not to populate petabyte-capacity datacenters but to ultimately improve the quality of life for the citizens of our earth.

Accomplishing this requires systems that process and analyze data to produce both real-time insights and insights over time that reveal patterns and trends. Those insights are what we, as human beings, then consume to drive decisions, investments, policy, and so on, as shown in Figure 1-1.

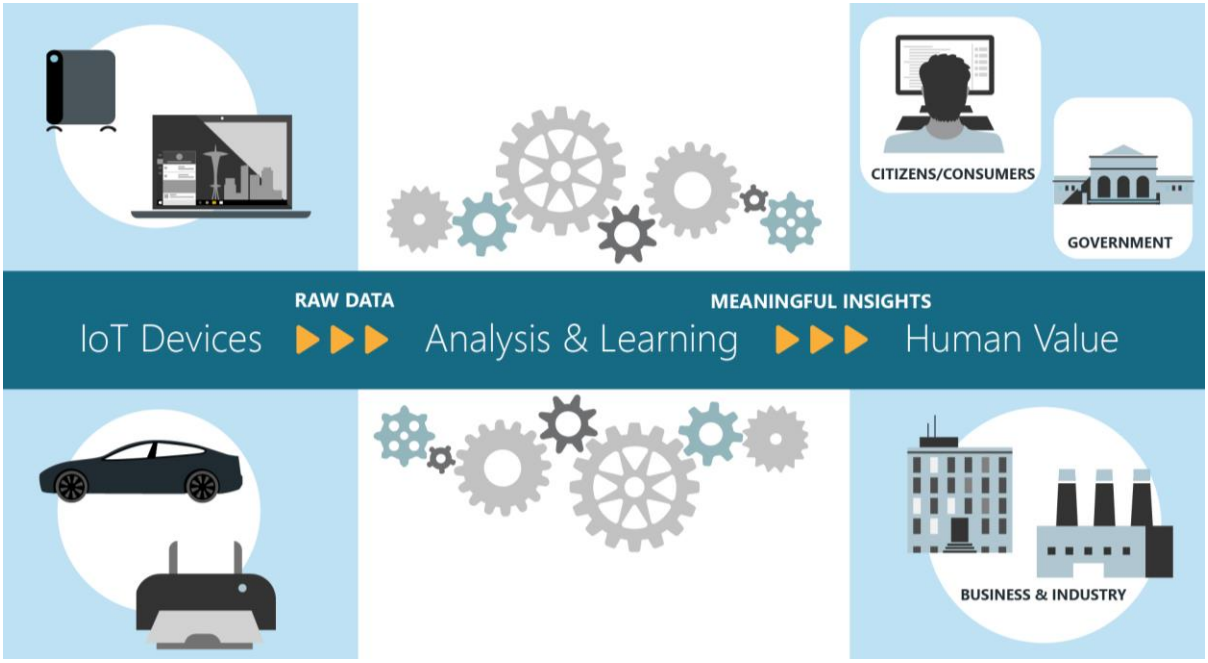


Figure 1-1: An IoT system applies analysis and learning to raw data to deliver meaningful value to humans.

* *IoT Devices to Almost Triple by 2020, to 28 Billion*, by Nathan Eddy, http://www.eweek.com/small-business/iot-devices-to-almost-triple-by-2020-to-38-billion.html, posted July 31, 2015.

The question is, how do you approach building such a system, especially if you're just getting started with IoT? That's what MyDriving is all about.

MyDriving provides a comprehensive starting point for a scalable, available, performant, and cross-platform IoT implementation. It brings together the best Azure, developer platform, and service offerings to demonstrate Microsoft's breadth and depth in this space: Azure IoT Hub, Stream Analytics, Machine Learning, Event Hubs, Service Fabric, SQL Server, HDInsight, and App Service, along with Xamarin, HockeyApp, Power BI, and Visual Studio Team Services. All the code is open-source on GitHub, and we've made it easy to provision a system of your own through the Azure Resource Manager template as explained in Getting started with MyDriving.

In this guide, we'll walk you through all the details of how MyDriving works, how it was built, and how to customize the system and begin delivering value to your customers in a variety of IoT scenarios.

> **Note** The design of MyDriving is based on the Azure IoT Reference Architecture, which is a helpful document to review when designing a similar system of your own.

The MyDriving consumer experience

To better understand what an IoT system can deliver, let's briefly explore how an individual consumer experiences MyDriving. For demonstrations, see the MyDriving videos from Microsoft //build 2016.

The human value that's delivered by MyDriving is a greater awareness of how your own driving habits affect safety and the long-term performance of your vehicle. Raw data is first collected from the vehicle's sensors in combination with your mobile phone. That data is then processed by the back end in real time to derive human value, which is then presented to you in meaningful and attractive ways on your phone.

> **Note** In this scenario, one mobile phone plays two distinct roles. It collects and transmits data as an IoT "field gateway," and it also provides data visualization/presenting value. We designed it this way because most drivers already have a phone. In general, though, these distinct roles can be filled by any number of other means, such as dedicated IoT devices and web or desktop applications.

To make MyDriving work, you need to connect the car and the mobile phone to the back end as follows and as shown in Figure 1-2 on the next page. The MyDriving user guide walks you through the details for these steps.

1. Buy an inexpensive dongle that plugs into the car's On-board Diagnostic (OBD) port and makes vehicle telemetry available over Bluetooth or Wi-Fi. Telemetry includes speed, distance, fuel consumption, temperature, RPMs, and more. (As described in Chapter 2, *IoT Devices,* we used the BAFX Products Bluetooth OBD dongle for Android and Windows Phone, and the ScanTool Wi-Fi OBD dongle for iOS.
2. Connect the dongle to your mobile phone via Bluetooth or Wi-Fi as you would with other such devices.
3. Install and run the MyDriving app on the mobile phone. As a field gateway, the app prompts you to log into the back end so it can associate private trip data with your user identity.

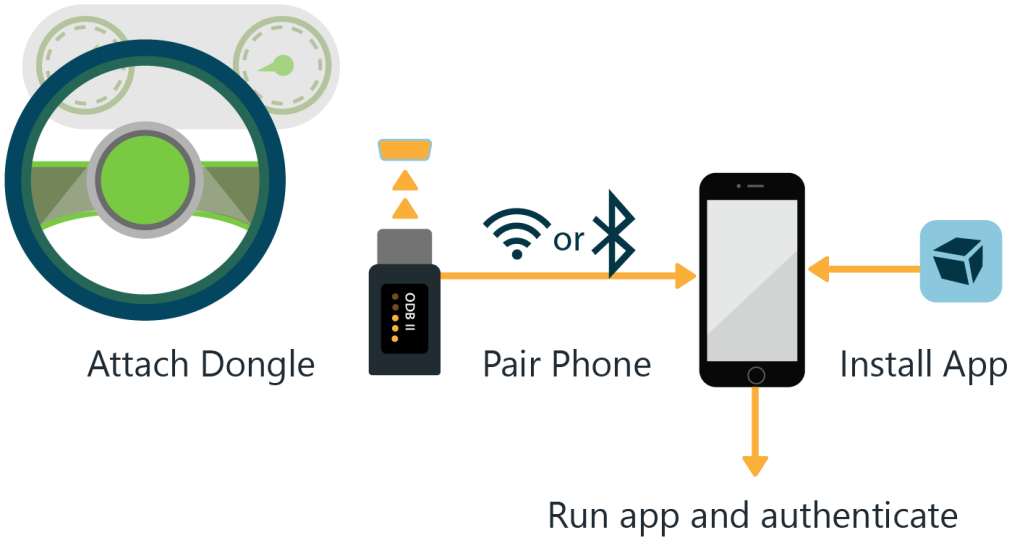


Figure 1-2: The connections necessary to experience the MyDriving system

With these connections made, you're ready to start recording a trip through the app's UI as shown in Figure 1-3.

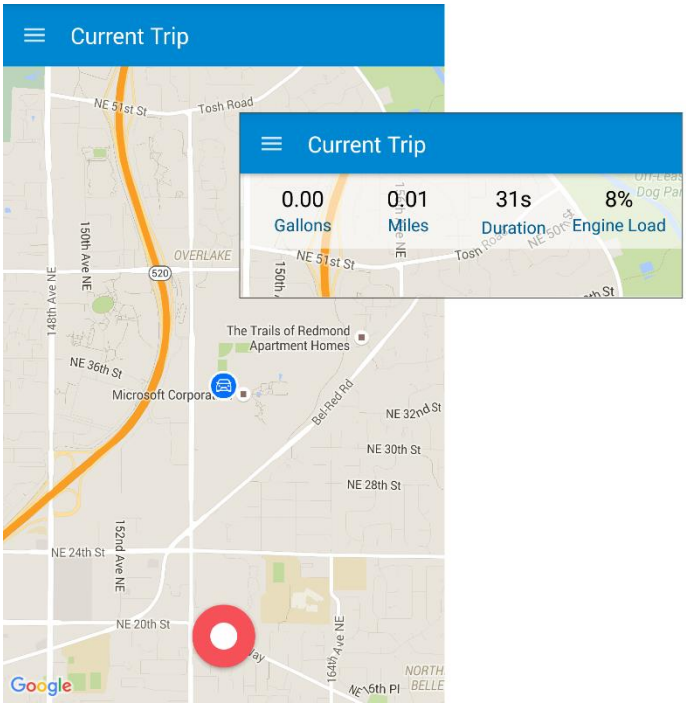


Figure 1-3: Recording a trip in the MyDriving app. The inset shows the overlay while the trip is underway.

During your trip, the app—again in the role of an IoT field gateway—collects data from the OBD device and from the phone's GPS sensor (Figure 1-4 on the next page). A dedicated (non-phone) IoT device does something similar, and might have a whole host of different sensors to draw from as well. In any case, the field gateway regularly uploads the data that's needed in the back end through the data connection it has to work with (Wi-Fi or cellular), using protocols such as HTTP, MQTT, or AMQPS. Depending on the IoT device's storage capabilities, it might also cache some data locally

that's not needed for real-time processing, which it then uploads in larger batches at a later time. The MyDriving app, for example, regularly sends OBD data to the back end while recording, but stores route information only locally. When recording is complete, it saves that route data to cloud storage.

Figure 1-4: The general data handling of the MyDriving app in its role as an IoT field gateway

When recording is complete, the MyDriving app stops being an IoT field gateway and assumes the role of data visualization, with which you can view and explore data for past trips, with route information overlaid on an interactive map (Figure 1-5).

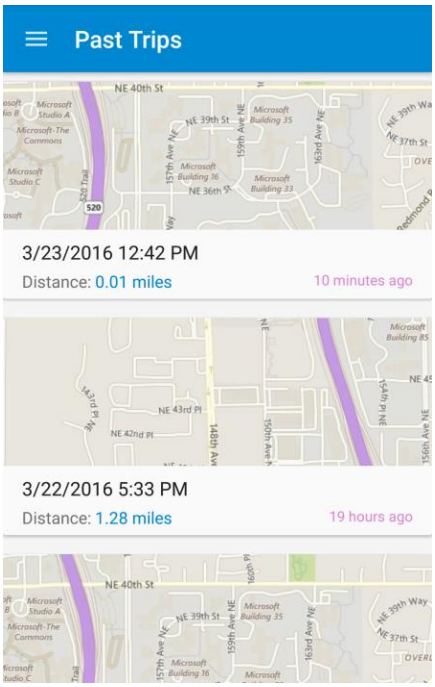


Figure 1-5: Viewing a past trip in the MyDriving app

The route also highlights "points of interest" where hard stops or hard accelerations occurred—behaviors that might indicate increased wear and tear on the car or potentially problematic driving habits.

The back end also combines this trip with prior trips to produce an overall driver rating, which you can see in the app through your user profile (Figure 1-6).

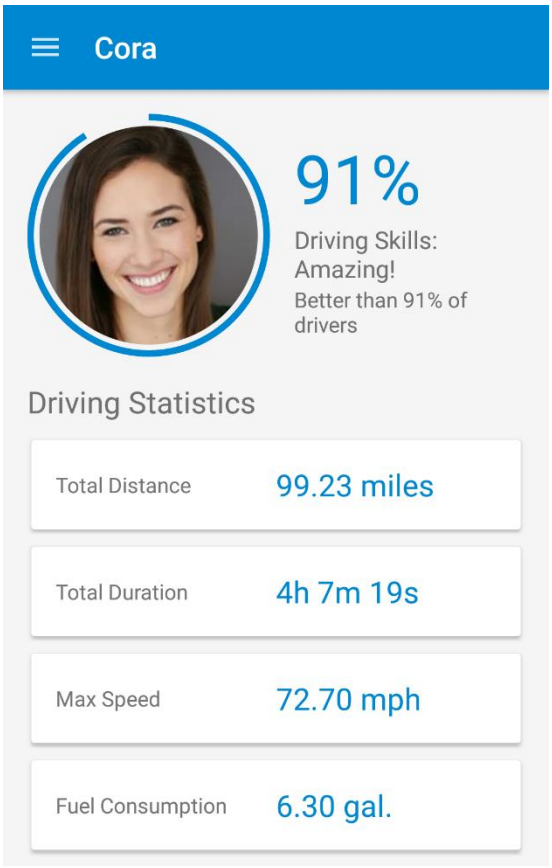


Figure 1-6: Viewing your profile with your driver rating in the MyDriving app

In the end, the data visualization is clean and simple because the goal here is *delivering value*: the awareness of your driving habits and where you might change behaviors to improve your overall rating.

In fact, with consumer-facing scenarios like MyDriving, we really don't want users to think about all the raw data and the processing power of the cloud that went into these apparently simple reports. Instead, we want them to quickly and clearly see what those results mean for them and their lives.

The MyDriving operator experience

In addition to providing an individual consumer experience, MyDriving also implements additional ways to visualize data *as a whole*. That is, while MyDriving delivers value to *individual* drivers through the app, it also delivers value in the form of the aggregate data *across* all drivers and devices. This is clearly of interest to the business that owns and maintains the system itself. The system's operators are likely to continuously monitor the real-time data that's coming in, and also to try to gain insights from the collection of historical data.

In MyDriving, the back end aggregates real-time and historical data together for these purposes, making it all available to Microsoft's suite of business and analytics tools called Power BI, as shown in Figure 1-7 and Figure 1-8, both on the next page.

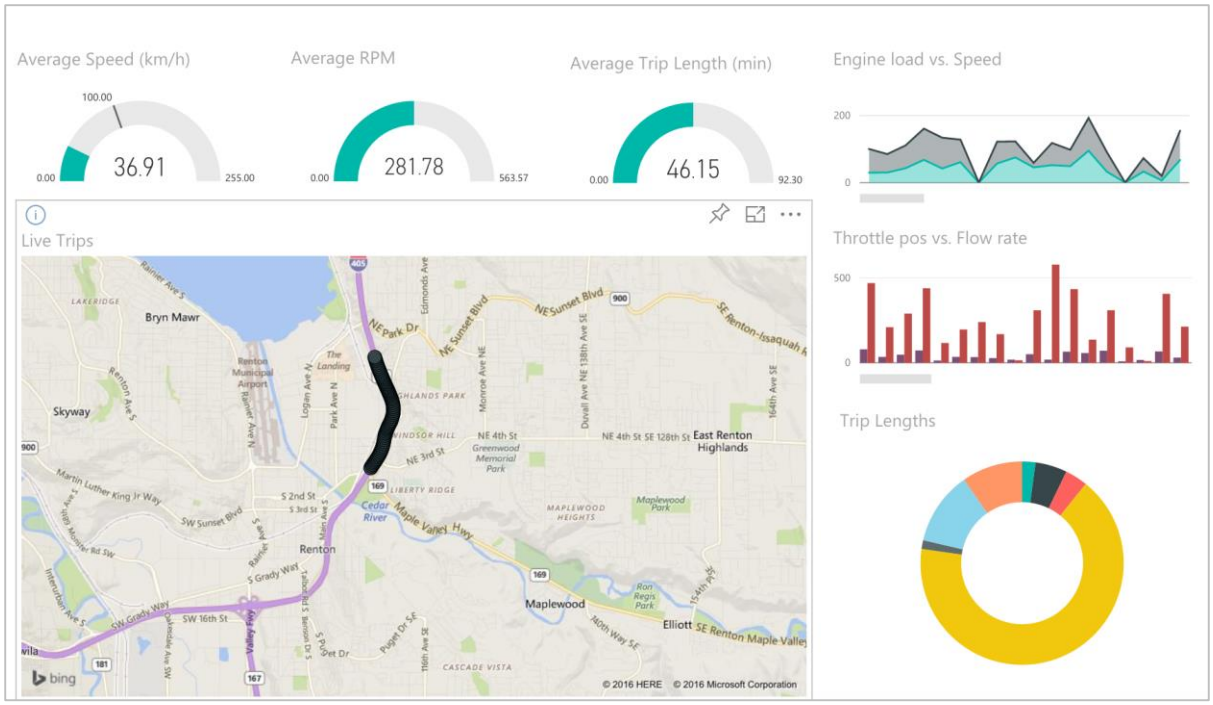


Figure 1-7: The Power BI experience for real-time data

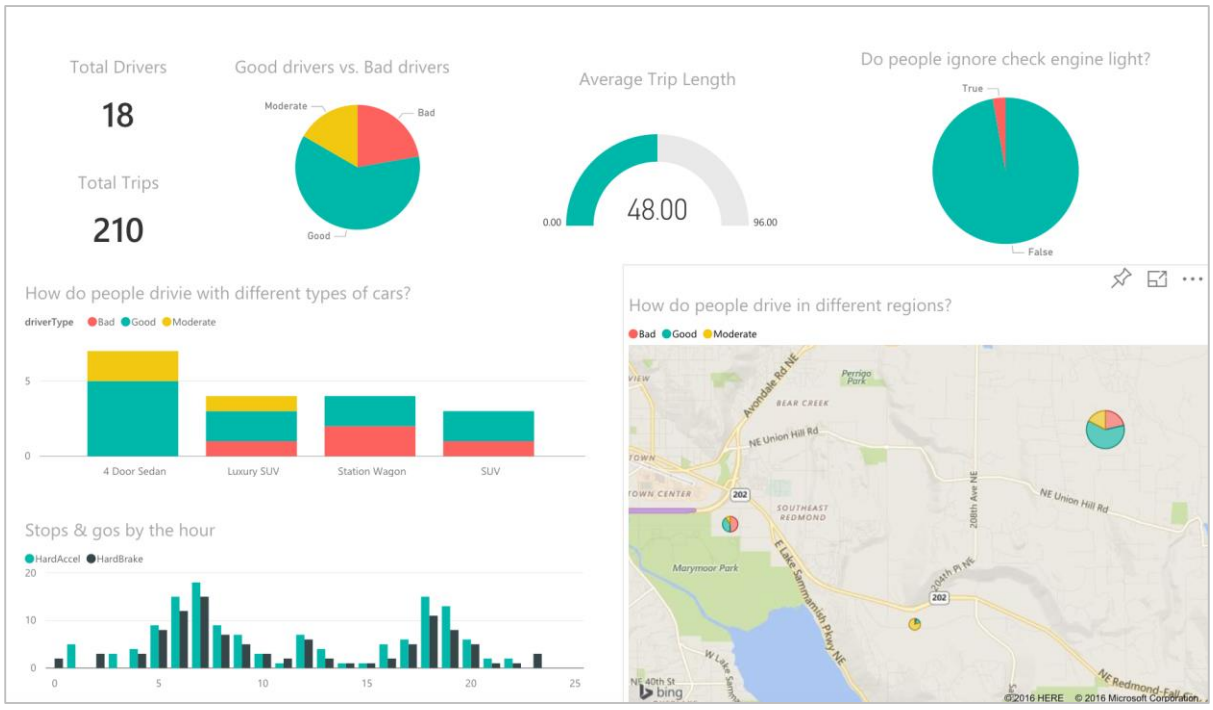


Figure 1-8: The Power BI experience for historical data

MyDriving as a starting point

The MyDriving scenario shows just a few possible ways to derive human value from a specific type of IoT data (that is, OBD and GPS data). There are clearly thousands if not millions of other possible scenarios, all with unique data and unique visualizations of the data's value. The IoT devices involved are likely to be very different across those scenarios, as befits the sensor requirements. The dashboards, apps, and other visualizations are likely to be unique as well.

Yet for all those variations and possible customizations, many IoT systems share certain characteristics as illustrated in Figure 1-9:

- Whether the data concerns weather, traffic, bird migration, economic activity, machinery lifecycles, epidemiology, or really anything else, each system has a scalable mechanism for collecting and storing data from a variable number of devices.
- No matter how technologically diverse those IoT devices are, the "Internet" part of the "Internet of Things" name means that they communicate their data through standard protocols like HTTP, MQTT, and AMQPS.
- Regardless of the rate of collection or the structure of the data, each system employs configurable back-end tools to analyze that data and learn from it over time. Each configuration expresses the highly specific rules and realities of the scenario in question, but the flow of applying that configuration to a data set is a common pattern.
- No matter how the insights from the analysis and learning might be ultimately be used, the back end makes its results available through standard mechanisms like HTTP or SQL databases. This enables developers to build any number of experiences around those results, including data visualization, control of automated systems (to reorder supplies, trigger service calls, and so on), and the ability to feed data into other systems entirely.

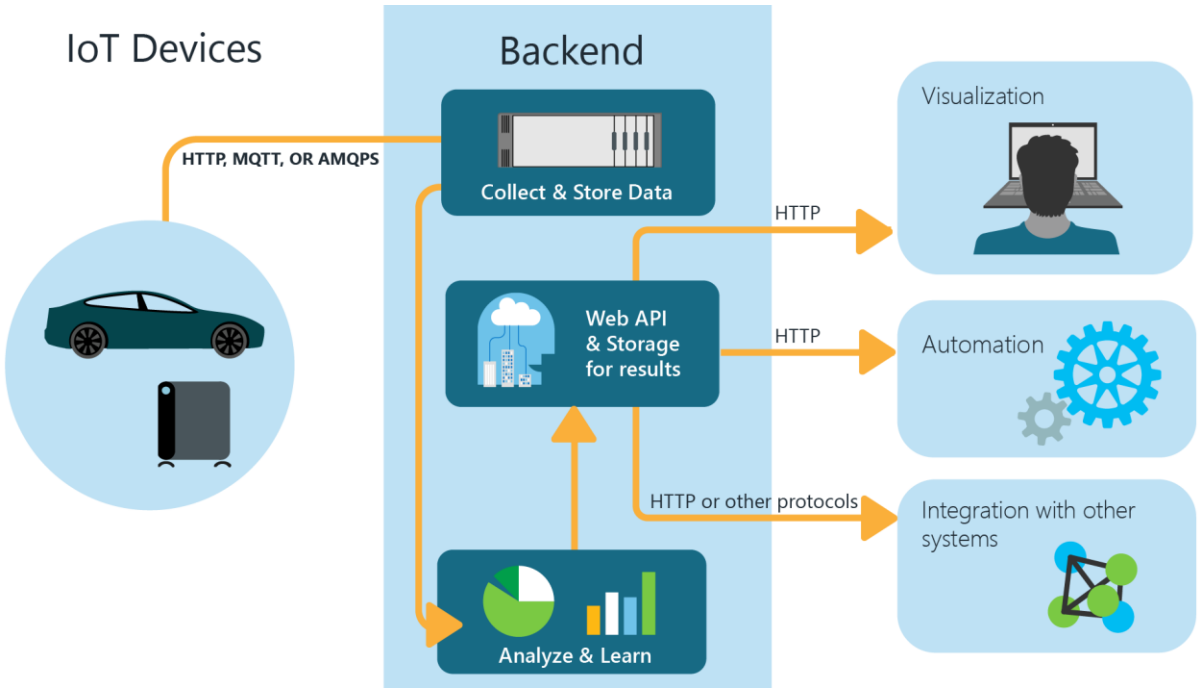


Figure 1-9: The general relationships within an IoT system

Given these commonalities, we've built the MyDriving project to serve as a starting point for deploying a solution of your own that can be readily customized for different scenarios. This helps you begin collecting and deriving value from IoT data more quickly and less expensively than if you started from scratch. And by using standard Azure services, you'll also have a lower cost of ongoing maintenance than if you build your own infrastructure.

By starting with MyDriving, in other words, you can focus on those aspects that are unique to your scenario—namely the configurations of analytics and machine learning, the API endpoints that make results available, and final user experiences that are built around those results. These, and not going through the process of building infrastructure, are what deliver unique value to your customers.

The MyDriving architecture and guiding principles

The overarching goal in the design of MyDriving is to bring together IoT devices, mobile apps, and Power BI through the orchestration of a rich cloud back end running on Azure. It also demonstrates the development workflows with a cross-platform app platform like Xamarin in conjunction with Visual Studio Team Services and HockeyApp.

Figure 1-10 gives you a complete view of the MyDriving architecture. The Azure services that you see here (the teal boxes) are what you'll get when you run the Azure Resource Manager template in your own Azure account as described in the Getting started guide.

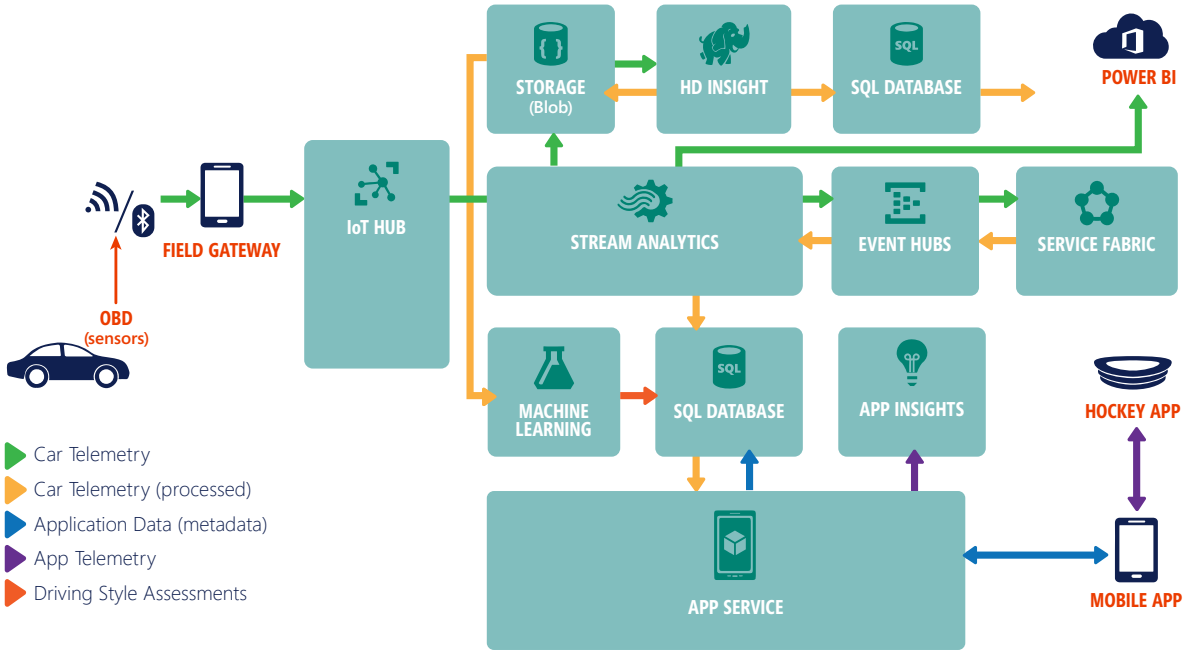


Figure 1-10: The overall MyDriving system architecture indicating the data flows and relationships

As you can see, this is a comprehensive architecture that employs many services with many relationships, which we'll learn about in this present guide. As such, it's adaptable to a variety of real-world scenarios.

The MyDriving demonstration, in fact, is really just one *instance* of this architecture, one in which OBD and GPS data makes a journey through the whole back-end system to ultimately present insights to human beings. By customizing different aspects, then, you can reuse most of the project to deliver value in other ways:

- You can use the same results from the back end for completely different purposes. For example, an insurance company might use driver ratings to offer discounts to improve safer driving habits and motivate drivers to change.
- You can customize the analytics and machine learning modules in the sample to derive entirely different insights from the same IoT data, around which you can build unique user experiences.
- You can change the type of data you're feeding into the system. This of course means customizing the back-end modules and the user experiences (as with the scenarios we just described), but you wouldn't need to rebuild the infrastructure.

The MyDriving architecture, in short, is not specific to automotive data. Rather, it provides a common pattern that's applicable to many different IoT + cloud + visualization scenarios: the transformation of large amounts of raw IoT data through analytics and machine learning to produce meaningful insights and user experiences.

Guiding principles

The design of MyDriving was influenced by a number of guiding principles as described in the following sections.

Inexpensive and purposeful IoT devices

Generally speaking, IoT devices should be simple and low cost, allowing for data collection from many individual points.

In the MyDriving system, we use a mobile phone as a field gateway to collect data from a non-networked device (the Bluetooth- or Wi-Fi-capable ODB dongle) and transmit it to the back end's cloud gateway. This choice makes sense in an automotive scenario because most drivers already have a mobile phone and wouldn't want to purchase a separate Internet-capable gateway device. They can plug into the scenario (literally) with an off-the-shelf ODB device.

In other scenarios, however, you can't assume that the consumer has a mobile phone. Alternately, a mobile phone might be more expensive than a custom IoT device built, for example, with a Raspberry Pi. Even with ODB data, there might be service fleet scenarios, for example, where you want to build dedicated IoT devices that can collect a richer set of data than we're doing with MyDriving, or that don't require direct interaction with drivers to record and transmit data.

What's important to understand, then, is that it's not mandatory to have a phone and an app serve in the field gateway role. The phone is merely connecting the ODB data source to the back end, and other IoT devices can be used for the same purpose.

Cross-platform mobile app experience

We wanted the mobile app to be available on all major platforms. This, along with the team's expertise in C#, were primary factors in choosing Xamarin as a cross-platform mobile technology. Xamarin is also well-supported in Visual Studio Team Services for running builds and tests in the cloud, including running automated tests within Xamarin Test Cloud. In MyDriving, we're also using HockeyApp for distribution of the app to testers and for telemetry and crash analysis.

The primary role of the app is to *present* data from the back end through a beautiful and engaging user interface. It need not, therefore, be overly concerned with *processing* of that data. This is a general principle for the design of cloud-connected mobile apps.

By their nature, mobile devices have limited power and occasional connectivity. The cloud, on the other hand, is always on and always connected, with enormous processing power at one's command. Thus, it's best to have the back end do the heavy lifting of data storage, analysis, and so on, and then make mobile-optimized data available through web API endpoints. In this way, we can minimize the network traffic (that is, data usage) from the mobile device and minimize its power requirements while still delivering a great experience.

Although apps often get all the credit for delivering a great experience, they're usually just providing an engaging viewport for the sophisticated processing and orchestrations that are happening in the cloud. The MyDriving app, in fact, isn't really part of the architecture *per se* because it's entirely oriented around the specific data and usage patterns involved. When you implement a scenario of your own based on MyDriving, you'll be replacing the client experience entirely.

Extensible Azure-powered back ends

Within the back end—the heart and soul of MyDriving—we sought to showcase how multiple Azure services combine into a meaningful whole without the need to write a lot of code or build services from scratch. We've built the back end primarily through the configuration and interconnection of services like Azure IoT Hub, Stream Analytics, SQL Database, HDInsight, Machine Learning, and App Services, along with external services like Power BI.

We also wanted an extensible design that you can customize for additional scenarios that aren't included with MyDriving. This customization happens primarily through Azure Event Hubs and Service Fabric (for microservices) as described in Chapter 9, *Microservice Extensions*. We provide an example of this in MyDriving with a microservice that does a VIN lookup for a vehicle. Note that when Azure Functions become available, you can also use them to implement extension capabilities.

To help you get started, we also made certain choices to keep the cost of running the Azure back end relatively low. For more information, see the section "Estimated operational costs" in the Getting started guide. In your own deployments, you might opt to invest in higher-performing services if your scenario requires them.

Short ramp ups and rapid iterations

One other characteristic of the design is that it was conceptualized, implemented, and refined over the course of about eight weeks. This proves that by bringing together many of Azure's platform-as-a-service (PaaS) offerings, it need not take 12-18 months to create a system similar to this—as it might if you built all the components yourself and deployed them to a stock virtual machine.

Indeed, with MyDriving as a starting point, we expect that you'll be able to deploy a customized system in as little as two to three weeks, with the ability to collect, analyze, and visualize the data that's available to you. This short ramp-up time means that you can get into an agile process almost immediately to continually improve, refine, and extend the system, and to deliver increasing value to your business.

Considerations for expanding and scaling the system

Although the MyDriving architecture is comprehensive, it does have its limitations both in its design and in the service levels created by the Resource Manager template. The list below outlines some of the areas you might consider when building a system of your own:

1. Establish the required service levels for Azure IoT Hub ingestion and the API endpoints in App Service based on the number of devices and number of consumers you're designing for.
2. Establish your tolerance for event-to-action latency, that is, the frequency of data collection on the device side. The MyDriving app, for example, collects OBD data from the vehicle once

per second or every 5 meters of movement, at which point it transmits that data to IoT Hub. Other scenarios would obviously have different sampling and/or batching requirements.

3. Select the matching IoT Hub plan based on the number of devices and the messaging rate you require.
4. Assess the Stream Analytics processing capacity in terms of the "Streaming Units," based on the required message throughput (which in turn is based on the rate of IoT data ingestion).
5. Modify the system architecture to logically separate storage for:

    - Device state (last known sensor values)
    - Application performance metrics (data that indicates whether the system is meeting business needs; for example, data ingestion rates)
    - Telemetry history (data that was previously transmitted to the cloud)

    Device-centric, point-in-time queries can be satisfied from the device state storage and/or telemetry history; aggregate queries can be met by the performance metrics storage. Analytics across device sensor values that use specific time windows are also possible here.
6. Define data-retention policies and create processes to export expired data into long-term storage such as Azure Data Lake. In the MyDriving architecture, we run a periodic (daily) process with HDInsight to remove data from the archive (blob) storage, shape it, and store it in a SQL database. If you need to retain that archive data after this step, then you'll need a separate process for copying it to long-term storage.
7. Plan for the required capacity in all storage components based on your data rate and data retention policies.
8. Define multi-tenancy strategy and design-appropriate mechanisms to isolate tenants at the application layer (for example, ingestion, stream processing, and API) as well as at the storage layer.
9. Define a business continuity plan based on the SLA and functionality commitment to your customers. This also involves deploying geo-distributed back-end services as needed.
10. Implement monetization-supporting instrumentation as well as throttling services.

In this context, be sure to review the Azure IoT reference architecture, upon which the MyDriving system is based. That document goes into much more detail about architectural options that will help guide your own implementations.

Roadmap of this documentation

Whereas the MyDriving Getting started guide walks you through setup and deployment steps, this present guide helps you understand how the architecture is put together and how we're using developer services like Xamarin, Power BI, HockeyApp, and Visual Studio Team Services. In the chapters that follow, we'll peel back successive layers to explore the system's components and their relationships, and where you can customize for your own scenario:

- Chapter 2, *IoT Devices*: the nature of IoT devices and their general operation.
- Chapter 3, *The Mobile App*: the details of the MyDriving app, written with Xamarin, in both its roles as an IoT field gateway and as a consumer app that provides data visualization.

- Chapter 4, *App Service API Endpoints*: the role of Azure App Service in MyDriving to provide API endpoints to provision IoT devices and to work with back-end data.
- Chapter 5, *DevOps*: how the app and back-end API projects are built with Visual Studio Team Services, deployed to HockeyApp (app) and Azure (API service), and monitored with HockeyApp (app) and Application Insights (API service)
- Chapter 6, *Real-Time Data Handling*: the path of IoT data through IoT Hub and Stream Analytics into storage for both the App Service APIs and Power BI.
- Chapter 7 *Machine Learning*: how Machine Learning is applied to trip data to produce additional results.
- Chapter 8, *Historical Data Handling*: the path of IoT data through Stream Analytics and HD Insight into storage used by Machine Learning and Power BI.
- Chapter 9, *Microservice Extensions*: the use of Event Hubs and Service Fabric to create extensions to the system, with a VIN lookup extension as an example.
- Chapter 10, *Reference*: details for data schema, data objects; example queries for Stream Analytics; and build and release definitions in Visual Studio Team Services.

When we introduce an Azure service (such as Machine Learning) or another technology (such as Xamarin) with which you might not be familiar, we provide a *primer* that gives you a short, general introduction to the technology in question. We then explain the specific *role* that the technology plays in the overall MyDriving system. In cases where we encountered *decision points* between different ways to fulfill a certain need, we provided you with information about the factors that drove our choices.

Through all this we hope to give you a comprehensive view of the MyDriving system at whatever level of detail suits your own role and responsibilities.

Feedback

Because we created MyDriving to help jumpstart your own IoT systems, we certainly want to hear from you about how well it works! Let us know if:

- You run into difficulties or challenges.
- You discover an extension point that would make it more suitable to your scenario.
- You find a more efficient way to accomplish certain needs.
- You have any other suggestions for improving MyDriving or this documentation.

Within the MyDriving app itself, you can use the built-in HockeyApp feedback mechanism—on iOS and Android just give your phone a shake—or use the **Feedback** menu command. You can also give feedback through the HockeyApp portal.

To give feedback on the rest of the system:

- File an issue on the GitHub repository.
- Leave a comment on the MyDriving Getting started guide.
- Leave a comment on the MyDriving post on the Azure blog.

We look forward to hearing from you!

IoT Devices

IoT devices, by design, are relatively simple. Their purpose is to collect raw data and send that data off to services that can do any number of interesting things with it. IoT devices can also receive commands from those services in return, allowing the results of processing in the back end to influence or control the operation of the devices. In this chapter, we'll examine the nature of IoT devices and the role they play in the MyDriving system.

IoT devices can be built in many different ways, as befits the "Things" part of the moniker. The basic requirement is that they communicate with the back-end cloud gateway through protocols such as HTTP, MQTT, or AMQPS. That's the "Internet" part.

In MyDriving, these "devices," as illustrated in Figure 2-1, are composed of an OBD dongle that's plugged directly into a vehicle, which talks to a mobile phone via Bluetooth or Wi-Fi, which in turn sends data to the back end via HTTP.

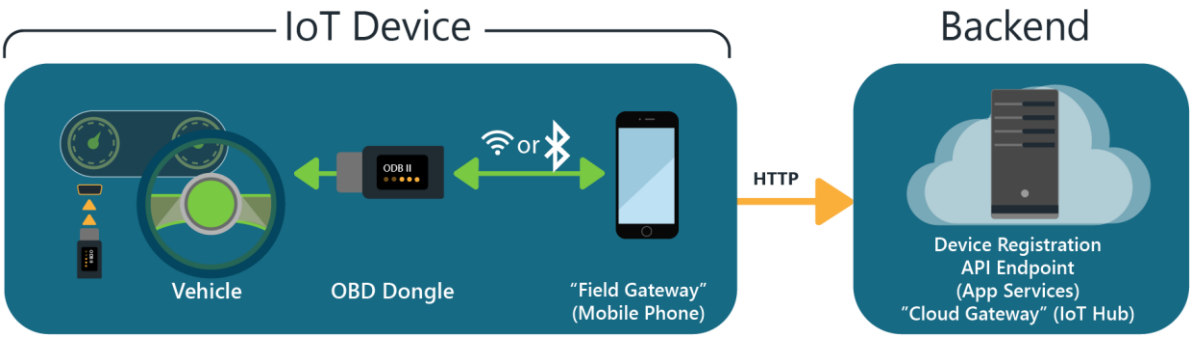


Figure 2-1: The nature of the IoT "device" within MyDriving

It's essential to understand that, in this role, the mobile phone is merely acting as a "field gateway." It receives and parses the low-level messages from the OBD dongle, packages that data along with GPS readings from the phone into JSON text, and routes those packages to the back end's cloud gateway (an Azure IoT hub in this case) via a supported protocol such as HTTP. This is the starting point for the flow of data throughout the system.

There's nothing special about using a phone here. We chose this model for MyDriving because most drivers already have a mobile phone and can acquire an inexpensive, off-the-shelf, OBD dongle to plug into their car. This avoids having to acquire a more expensive dedicated IoT device, which is to say, minimizes the overall cost of the user experience.

In your own scenarios, on the other hand, a mobile device might not be appropriate or assumed. In that case, building a dedicated device with a Raspberry Pi or custom OBD device might be more cost-effective. Again, it doesn't matter so long as the "device" here can interact with the cloud gateway via one of its supported protocols.

In this chapter, we'll explore the details of IoT devices, starting first with some background on IoT devices and field gateways. Then we'll see how they manifest in the MyDriving architecture through the decisions that led to our choices.

The details on how the app actually collects and transmits data, though, we'll save for Chapter 3, *The Mobile App,* because that process is best discussed in the context of the overall app.

Primer: what is an IoT device?

An IoT device is a physical device that collects data from one or more sensors and shares that data with a cloud-based service. For example, a simple device might use a sensor to collect the temperature in the environment and send that value once per second to a cloud-based monitoring system.

In *command-and-control* scenarios, IoT devices can also receive and act on commands that are sent from the back end. For example, a cloud-based monitoring system might send a command to a device telling it to open a valve.

The following is a list of scenarios where IoT devices might be used, or types of technologies they might be used with:

- Trip tracking and car health
- Sports and fitness tracking
- Health monitoring
- Person, pet, or livestock tracking
- Smart appliances
- Home automation and security monitoring
- Beacons or proximity sensors
- Smart vending machines
- Environmental monitoring such as street-level pollution sensors
- Asset tracking
- Industrial automation controllers
- Industrial equipment monitoring for predictive maintenance
- Manufacturing process monitoring devices

As you can see from this list, there are a variety of IoT scenarios in which devices need to collect and share data, and others in which the devices need to act on commands that are sent from a service. We can identify a number of device characteristics, which are described in the following table. The exact nature of the devices will, of course, depend on the specific usage scenario.

Device characteristic	Example
Often embedded systems with no human operator.	A smart vending machine tracks stock levels and automatically requests refills.
Typically a special-purpose device.	An IoT device (unlike a phone or tablet) usually has a specific function, such as reporting the temperature in the environment.

Can be used in remote locations where physical access is very expensive.	A sensor attached to a pipe in a remote oil pumping installation.
Might only be reachable through the solution back end.	An aircraft engine monitoring device may be reachable only from the monitoring service.
Might have limited power and processing resources.	A health monitoring band worn on the wrist.
Might have intermittent, slow, or expensive network connectivity.	A device in a car has no network access when there is no cellular coverage or when the car is in a tunnel.
Might need to use proprietary, custom, or industry-specific application protocols.	Cars typically expose on-board diagnostics by using the OBD-II protocol. Industrial automation controllers might use protocols such as DeviceNet, PROFIBUS-DP, or CAN.
Might be created by using many popular hardware and software platforms.	Raspberry Pi, Arduino, or BeagleBone devices.
Might only send data to a service or only receive data from a service.	A car on-board diagnostics system sends telemetry only to the back-end system; a home automation system reports information about the home and also enables users to remotely control lights and temperature.
Might send or receive sensitive data that requires a secure communication channel.	A person-tracking system for children should allow only parents or other designated individuals to access information about a child's location.

Primer: what is a field gateway?

Many IoT solutions include a *field gateway* device that sits between IoT devices and the services with which they communicate. A field gateway is typically located close to those devices. Field gateways are often used to enable connectivity and protocol translation for devices that either cannot or should not connect directly to the Internet, such as devices that support only Bluetooth (cannot connect) and devices that are unable to use a secure protocol when using a public network (should not connect).

A field gateway differs from a simple traffic routing device (such as a network address translation device or firewall) because it typically performs an active role in managing access and information flow in the solution. A field gateway might do any of the following:

- Manage local devices. For example, a field gateway can perform event rule processing and send commands to devices in response to specific telemetry data.
- Filter or aggregate telemetry data before it forwards it to the service. This can reduce the amount of data that is sent to the service and potentially reduce costs in the solution. In the MyDriving solution, the phone sends a snapshot of the OBD data every second (or 5 meters of movement) rather than continuously streaming data from the OBD dongle.
- Help to provision devices, that is configure them to work with the cloud gateway.
- Transform telemetry data to facilitate processing in the back end. In the MyDriving solution, the phone formats the OBD data as JSON before sending it to Azure IoT hub.
- Perform protocol translation to enable devices to communicate with the back end when the devices do not use the transport protocols that the back end supports. In the MyDriving solution, the phone translates from the low-level protocol used to communicate with the OBD dongle to HTTP to communicate with Azure IoT hub.

We often characterize field gateways as being *transparent* or *opaque*. A transparent gateway (Figure 2-2) forwards messages to the back end, leaving the original device ID intact so that the service is aware of all the connected IoT devices.

An opaque gateway (Figure 2-3) forwards data to the back end using its own ID, so that the service is aware of only the field gateway as a connected device. Note that although you typically deploy a field gateway that is local to your devices, in some scenarios you might deploy a protocol translation gateway in the cloud.

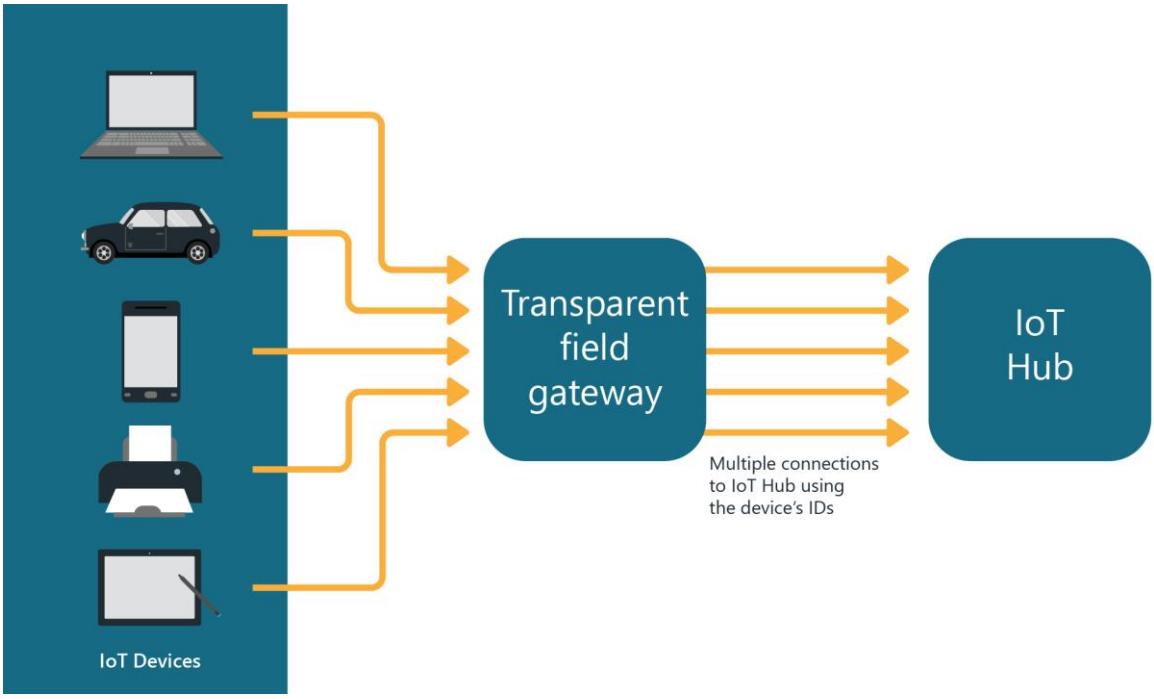


Figure 2-2: A transparent field gateway forwards messages to the back end without changing device IDs

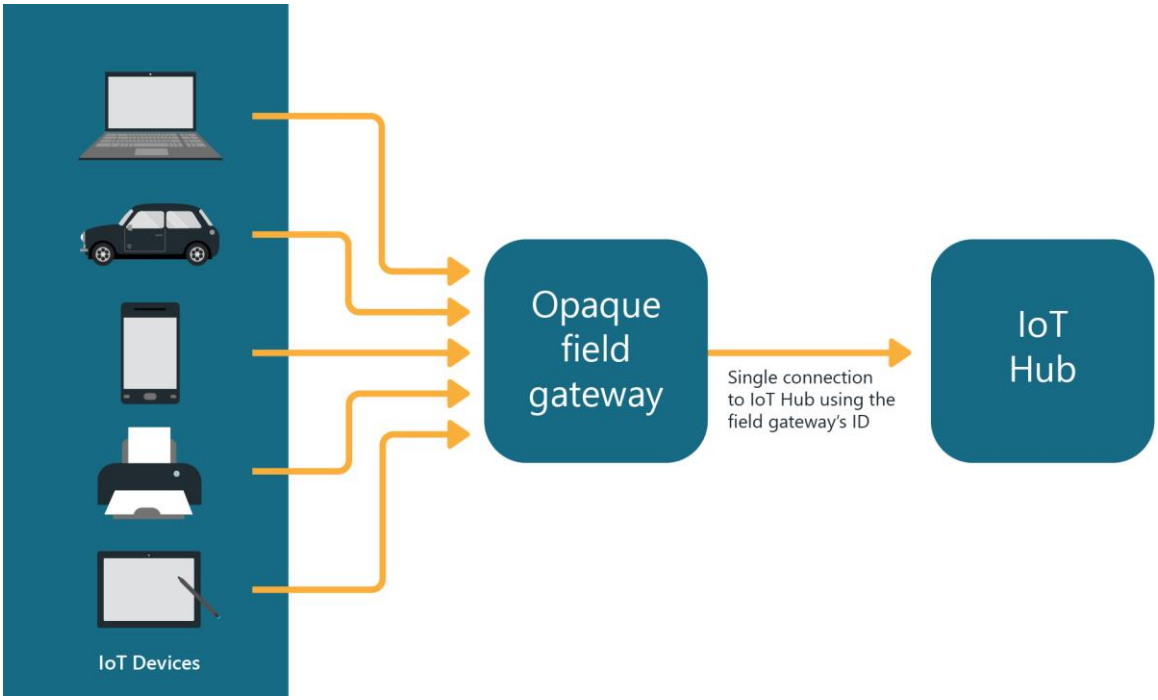


Figure 2-3: An opaque field gateway forwards messages to the back end with its own ID.

Decision point: OBD devices

The MyDriving solution uses On-board diagnostics (OBD) data from your car to analyze your trips and your driving. Modern cars have a standard OBD-II Data Link Connector somewhere in the cabin, so by plugging an OBD-II-compliant dongle into this connector, OBD vehicle data can be made available to other local devices via USB, Bluetooth, or Wi-Fi.

Depending on the make, model, and age of your car, you can collect various types of OBD telemetry such as speed, fuel efficiency, and engine RPM. You can also read additional diagnostic data associated with the "Check engine light" indicator on the dashboard that can help identify the specific component that has malfunctioned.

> **Note** The wireless connection used by an OBD device might not be secure, meaning your OBD data could be exposed to others. In addition to retrieving potentially sensitive and personally-identifiable information (PII) from the car, an OBD dongle plugged into the OBD-II Data Link Connector can also make changes to the car's firmware.
>
> Because of this, any solution that accesses a car through the OBD-II connector must take appropriate steps to prevent unauthorized access to the OBD connection. MyDriving, for example, requires user authentication for the app and does not use a reverse connection to the device from the back end.
>
> In IoT scenarios it's generally best to assume that every component in the system is a potential entry point for malicious hackers, and should thus be reviewed in the context of appropriate threat models.

In MyDriving, we use a number of OBD values, including speed, engine RPM, throttle position, fuel rate, and more. See *OBD data schema* in Chapter 10, *Reference*, for the exact values parsed from OBD

messages, and *Application data schema/TripPoint* in that same chapter for the data that's recorded for points along a trip route.

During project development, we tested several different OBD-II dongles to collect this data as described in the following table. Most OBD dongles are not Internet-capable, and thus require a field gateway to connect to the back end. Note also that there are many other OBD-II compliant dongles to choose from, with a range of features and prices. For this project, we focused on those that are relatively inexpensive and can still provide the necessary telemetry.

Device	Notes
Bluetooth OBD dongle for Android and Windows Phone	Android phones can connect to only a single IP network at any given time. Therefore, if the phone connects to the OBD dongle over Wi-Fi, it cannot simultaneously connect to the cloud-based back end by using the cellular data connection. For this reason, we use a Bluetooth dongle with Android phones.
Wi-Fi OBD dongle for iOS	iOS certification requirements mean that many Bluetooth dongles cannot be used with iOS devices. For this reason, we use a Wi-Fi dongle with iOS phones.
OBD emulator	During the development process, we found it convenient to work with an emulator that generates OBD data. This enabled us to develop and debug the code at our desks rather than sitting in a car. You can also use the MyDriving app with simulated data.

Decision point: field gateways

Because most OBD dongles cannot connect directly to the Internet, but work only over local connections such as USB, Bluetooth, or Wi-Fi, we needed another device in the MyDriving solution to act as a field gateway. In this role, the device must be able to use one of the secure protocols supported by IoT Hub as shown in the table below. Note that these are all secure protocols that encrypt the data that the device exchanges with IoT Hub. If your IoT device cannot use one of these protocols, you must use a protocol translation gateway.

Protocol	Port(s)
HTTPS	443
AMQPS	5671
AMQPS over WebSockets	443
MQTT	8883

Again, in MyDriving, we chose to use a mobile phone as the field gateway because most drivers already have a phone that can make ad-hoc local connections by using Bluetooth or Wi-Fi, so it's an appropriate place to deliver the consumer experience. By installing an app on the phone, a driver can easily and cheaply make the phone operate as field gateway.

It's certainly possible to connect some OBD devices directly to the Internet without a phone. The Freematics ONE OBD device, for example, has an xBee socket for connecting wireless communications modules that use GSM or Wi-Fi.

This particular device can also connect to a GPS receiver and has various on-board sensors such as a gyroscope and accelerometer that enable you to capture a richer set of data. It's also compatible with an Arduino UNO, which means you can program the device directly and customize the OBD and sensor data it sends to IoT Hub.

In any case, the phone in MyDriving acts as an opaque gateway because the OBD device does not have a unique ID assigned by the IoT hub in the back end. The MyDriving app uses its own device ID instead.

Also, the phone is not acting merely as a protocol converter that translates OBD data and relays it to IoT Hub. The app collects data from its own GPS sensor that it merges with the OBD data before sending it to the cloud, and also caches local data for later visualization purposes.

> **Note** If you don't have an ODB device, the MyDriving app will use simulated OBD data instead, sending that to the back end along with real GPS data.

The OBD interface in the car provides read-only data (unless you are a mechanic in a garage resetting a warning light after making a repair), which limits the scope for extending the MyDriving architecture to include command-and-control scenarios. However, the phone can respond to a command from the back end to implement such capabilities. For example, if the data sent from the OBD device indicates that the car is braking sharply, the back end can recognize this and instruct the phone to activate a dashcam to record what's happening.

Choosing a protocol for use with IoT Hub

With a phone and app serving as a field gateway, we're ready to see how these roles are fulfilled in the app itself. To do that, we need to look into the MyDriving app, which is the subject of Chapter 3. If you want get right to the details of sending data to IoT Hub, skip ahead to the section "App flow: IoT field gateway" in that chapter.

Simply said, the app connects to the OBD device when the user begins recording a trip, and during recording it polls data from the OBD device, combines it with its own GPS data, and sends it to IoT Hub over HTTP using the Microsoft Azure Devices Client PCL.

As noted before, IoT Hub supports HTTP, MQTT, and AMQPS protocols. There are a number of key considerations in choosing the one you'll use:

- **Library support:** The Microsoft Azure IoT SDK GitHub repository contains C, .NET, Java, and Node.js libraries to enable devices to work with IoT Hub, but not all libraries support all protocols.
- **Cloud-to-device pattern:** HTTP does not have an efficient way to implement server push. As such, when using HTTP, devices must poll IoT Hub for cloud-to-device messages. This is very inefficient for both the device and IoT Hub, and introduces latency in command delivery to a device. Although MyDriving does not send commands to the device, a possible extension is to send a command to switch on a dashcam if the back end detects sudden breaking. To minimize latency in delivering commands, use the AMQPS or MQTT protocol.
- **Payload size**: AMQPS and MQTT are binary protocols, which are significantly more compact than HTTP. Using AMQPS or MQTT helps minimize any charges that arise from the phone's cellular data connection to IoT Hub.
- **Field gateways**: When using HTTP or MQTT, you cannot connect multiple devices (each with its own per-device credentials) using the same transport layer security (TLS) connection. It follows that these protocols are suboptimal when implementing a field gateway because they require one TLS connection between the field gateway and IoT Hub for each device that's connected to the gateway. However, in the current solution there is only one OBD dongle per phone, so there is only one TLS connection to IoT Hub.

- **Low resource devices**: The MQTT and HTTP libraries have a smaller footprint than the AMQP libraries. As such, if the device has few resources (for example, less than 1 MB of RAM), these protocols might be the only protocol implementation available. However, modern smart phones typically have sufficient RAM to use the AMQPS protocol. (Note that the C library that supports the AMQPS protocol is now much more compact than earlier versions.)
- **Network traversal**: MQTT uses port 8883. This can cause problems in networks that are closed to non-HTTP protocols. You can use both HTTPS and AMQPS over WebSockets in this scenario. However, this is unlikely to be an issue over the public data network that's used by the smart phone.

Only the first consideration that's described here is relevant in the MyDriving solution. Again, we're using the Microsoft Azure Devices Client PCL, which was actually built in the process of creating this demonstration to provide a PCL with HTTP support across all platforms. We'll see more about how this library is used in Chapter 3.

CHAPTER 3

The Mobile App

Having explored the IoT ingestion side of the MyDriving system in Chapter 2, *IoT Devices*, we can now jump over to the other side, where the value that's derived from IoT data is delivered through a user experience. In MyDriving, the experience for an individual driver happens through the same mobile app that is used to record and transmit IoT data. In this chapter, we'll look at that app, which is implemented with Xamarin, and pay especially close attention to how it fulfills its roles as both an IoT gateway and for data visualization.

Generally speaking, any value that's delivered through a sophisticated back end like that of MyDriving happens through mechanisms such as API endpoints or the ability to access storage like the SQL database that's shown in Figure 3-1. With this type of loose coupling, you can build any number of user experiences around those results. You can also use the results without any user experience at all, as when feeding them into automated systems.

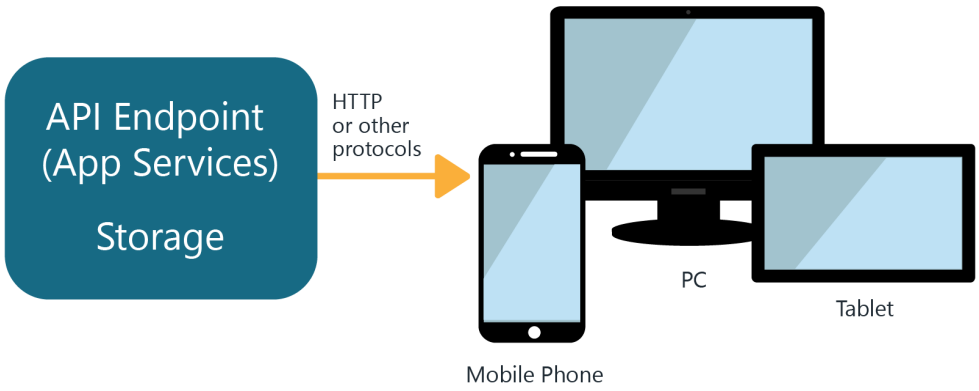


Figure 3-1: Results from the back end are delivered through API endpoints or storage mechanisms without restrictions on how they might be used.

In MyDriving, we have two routes for data visualization. First, as explained in this chapter, we use Xamarin to deliver apps on iOS, Android, and Windows, through which an individual user interacts with his or her personal data. Similar experiences can also be delivered through web browsers or other cross-platform app technologies like Apache Cordova. The second route is data visualization *across* users and devices, which is accomplished in MyDriving through Power BI and its ability to connect to back-end data stores directly. We'll learn more about Power BI starting in Chapter 6, *Real-Time Data Handling*.

For the mobile app, we'll start by looking at some of the options for cross-platform development and the reasons we decided to use Xamarin with native UI layers. We'll then delve into the structure of the MobileApps solution that's in the GitHub repository, doing a detailed walkthrough of the main app flows: authentication, IoT field gateway, and data visualization. Again, the latter two roles are separate

and can be fulfilled by different components when appropriate; they just happen to be in the same app in the MyDriving scenario.

Decision point: choices for cross-platform development

However you might slice and dice today's market for personal computing devices, the reality is that developers today must serve customers on multiple platforms. Android is clearly the most widely-used mobile platform in the world, and iOS is also very strong as the platform that generates the highest revenue for developers. Windows serves mobile customers too and has a very strong presence on the desktop and in the console market and is expanding into new areas.

Although it's possible to develop a native app for each platform individually and deliver a great user experience, the costs of doing so can be prohibitive, both in terms of time to market and total cost of ownership across the app's lifetime. To help control and lower these costs, different cross-platform development technologies have evolved to produce platform-specific app packages from a shared code base. The table below describes a few of the most popular ones for general-purpose apps.

Technology	Description
Xamarin	Native app packages are built with C# and .NET with components that provide full access to platform APIs. This results in a high degree of shared code between platforms, including the option to use Xamarin.Forms for shared UI code. Xamarin produces platform-specific app packages.
Apache Cordova	Hybrid apps are built with HTML, CSS, and JavaScript, and run inside a webview control within a native app wrapper (or run native on platforms like Windows). Plug-ins provide access to platform APIs from shared code. Apache Cordova also produces app packages.
Mobile web	Apps are built with HTML, CSS, and JavaScript to run in a mobile browser. They are deployed to a web host rather than appearing in platform app stores. Access to platform APIs is limited to the APIs that are exposed through HTML5.

Note that there are a number of other technologies in the marketplace for cross-platforms apps. These include technologies that are oriented toward high-performance gaming and graphics, such as Unity and CoCos2D. These can certainly be used for data visualization if they're appropriate for your own scenario.

The technology you'll choose for your own projects depends greatly on the skills that your development team has or can readily learn, the costs involved, and whether the technology itself is suitable for the user experience you want to create. Clearly, there wouldn't be so many options if one solution fit every team and every scenario! The fact is that each technology can produce great user experiences in the right circumstances and poor user experiences when used inappropriately or with sloppy engineering.

For MyDriving, we chose to use Xamarin because our teams already had strong C# and .NET experience, and because it allowed us to create beautiful, high-performance apps with native UI layers for each platform. That choice is described in the next section.

Learn about Xamarin: For more information about Xamarin, visit http://www.xamarin.com. For a walkthrough on setting up Xamarin with Visual Studio, see Visual Studio and Xamarin on MSDN.

Decision point: Xamarin.Forms or native UI layers

Xamarin provides two ways to build great native apps with a high percentage of shared code across platforms: "traditional" Xamarin with native UI layers and Xamarin.Forms.

With a traditional Xamarin approach, you write separate UI code for each target platform—iOS, Android, and Windows—by using the native designers (which are integrated into Visual Studio). This gives you direct access to native UI controls, which means you can create an optimized UI experience. You also have full access to the native controls for each platform to help with building the respective UIs. With this approach, developers typically realize between 50-80% shared code across platforms, depending on the nature of the app.

Xamarin.Forms provides a generalized API that lets you write a single shared UI layer for all platforms using C# and XAML. At runtime, Xamarin.Forms renders native controls, resulting in a native look and feel. That is, Xamarin.Forms *does* produce fully-native UI through abstractions that let developers realize nearly 100% shared code across platforms. With Xamarin.Forms, you can also take advantage of platform-specific features using Xamarin.Forms dependency services and custom renderers.

Deciding between traditional Xamarin and Xamarin.Forms often comes down to what's most important to you: pixel-perfect UI control, or more code sharing. That said, you don't need to decide which approach to take up front; apps can be implemented using a combination of both. For most projects, we generally recommend starting with a Xamarin.Forms solution to set up UI code-sharing across platforms. Then, on an as-needed basis, you can add platform-specific details using dependency services or custom renderers, or create specific screens using traditional Xamarin. Xamarin.Forms is especially helpful when your development staff does not have native UI experience on the target platforms. They can learn Xamarin.Forms and apply that knowledge across platforms.

Our choice: With the UX requirements in the mobile app of MyDriving, we could have chosen either path but we chose to use native UI from the beginning for the following reasons:

- MyDriving heavily uses around geolocation and maps. Currently it's easier to work with the full extent of mapping capabilities, such as overlays, within a native UI than it is within Xamarin.Forms. Although Xamarin.Forms has a map control, it's currently better suited to simpler uses such as displaying pins and points of interest.
- The other screens in the app are relatively simple to implement with native UI, and our team already had the necessary expertise. The ease of working with overlays in the native map controls made up for the small extra effort that was required to implement native UI for these screens.
- Native UI provides more precise control over fine details such as animations and transitions. This meant we could polish the UX to a greater extent than Xamarin.Forms presently allows.
- Native UI gave our staff the opportunity to use the visual designers for iOS, Android, and Windows to create the best user interfaces for each platform.

As a counter-example, members of the MyDriving team also built the app for the Xamarin Evolve 2016 conference for which they chose Xamarin.Forms. For a short-lived, simple event app, it was much more important to build the app quickly with nearly 100% shared code, and to keep costs low, than it was to create an optimized UI. The app's UI is oriented around presenting data from a cloud back end—a requirement that's easily satisfied with Xamarin.Forms.

Project structure

Now let's take a look at the Xamarin app solution to understand how it's structured to play both the role of an IoT field gateway and that of a consumer app. To follow along, download the code from the src/MobileApp folder in the GitHub repository, and then open the **MyDriving.sln** solution in Visual Studio.

> **Note** There are two other solution files in that folder: MyDriving.XS.sln is for use with Xamarin Studio, and MyDriving.iOS.sln is used by the build definition in Visual Studio Team Services (see Chapter 5, *DevOps*).

Figure 3-2 shows what you'll see in Solution Explorer if you expand the folders and reveal the individual projects. Note that the Utilities/Obd* projects are located in the repository in src/ObdLibrary, which is separate from the MobileApp folder.

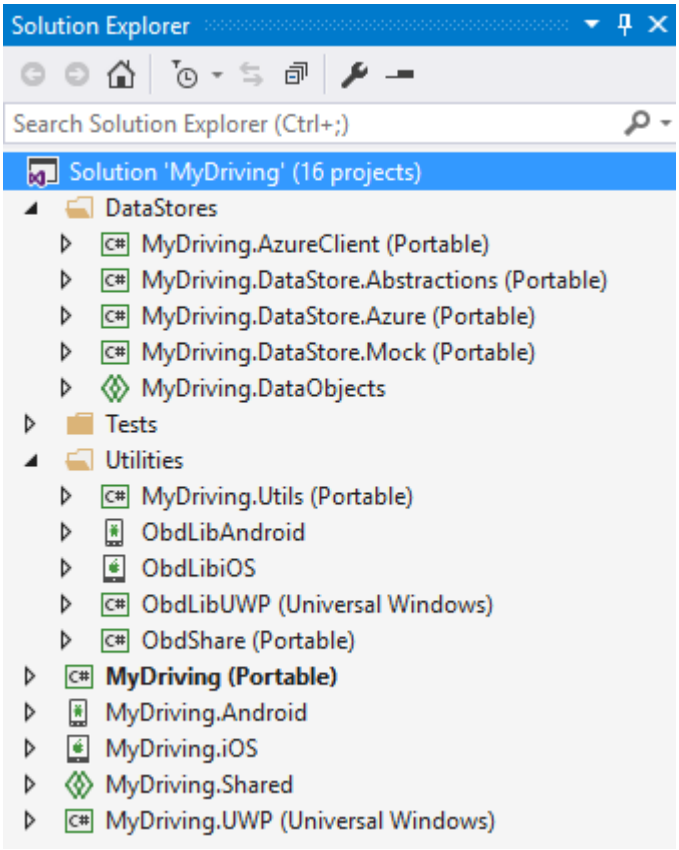


Figure 3-2: Projects in the MyDriving solution in Visual Studio

Figure 3-3 (on the next page) shows how all the distinct projects in this solution flow into the app packages we need for each platform's marketplace. These can be built locally or in the cloud with Visual Studio Team Services, as described in Chapter 5.

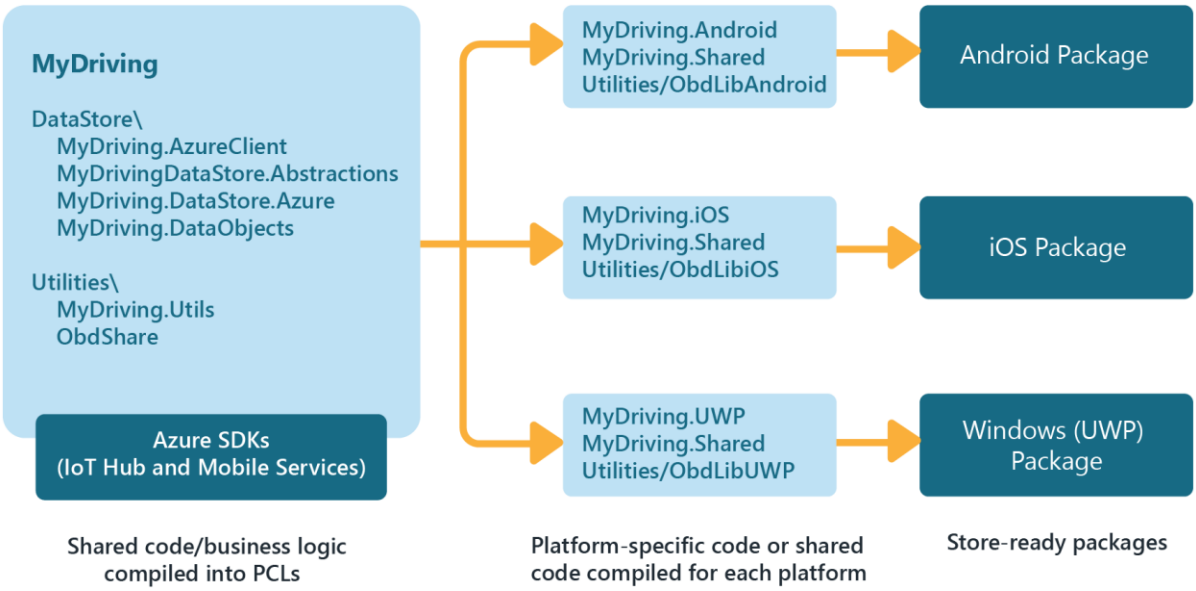


Figure 3-3: How the projects in the MyDriving solution flow into app packages

The projects on the left (along with MyDriving.DataStore.Mocks, which is used for testing in place of MyDriving.DataStore.Azure, but which isn't shown here), make up the body of code that's shared across platforms. Most of these projects build a portable class library (PCL) that is compiled once and then directly bundled into each platform-specific app package. The one exception is the MyDriving.-DataObjects "shared" project. Its code is pulled directly into both MyDriving.DataStore.Abstractions and the back-end App Service API project (MobileAppService in the repository, see Chapter 4, *App Service API Endpoints*) during their respective build processes, keeping both in sync.

> **Note** For a complete introduction to portable class libraries, see Cross-Platform Development with the Portable Class Library on MSDN.

The bulk of shared code is in the MyDriving PCL. In this PCL, you'll also find the view models that are used within standard MVVM structures throughout the app. (If you use Xamarin.Forms instead of native UI, we generally recommend that you have one PCL like this with data models, business logic, and MVVM bits, and another PCL with the shared UI code. This maintains a separation of concerns between views and view models.)

These view models implement most of the app's behavior, leaving the platform-specific UI in those projects to display that behavior. Thus the three platform-targeted projects in the middle of Figure 3.3 each contain assets like graphics, app manifests, UI definitions (views), and UI-related code (C#) that's needed for only that one platform. All the views in these projects plug into the MyDriving PCL view models or use standard data-binding mechanisms.

You can also see in Figure 3.3 that the code in the MyDriving.Shared project is brought into each platform package directly. An interesting point is that MyDriving.Shared is the only place you'll find any platform-related `#if` directives in the entire solution. This is because code in the platform-target projects has no need for such directives. Also, because the PCLs are compiled and included as binaries in each app package, they're platform-neutral by definition.

A few more notes about the projects:

- The Utilities/ObdLib* projects build platform-specific libraries for working with OBD, with Utilities/ObdShare containing bits of platform-neutral code. All these clearly relate to the

app's role as an IoT field gateway; again, the code is in the src/ObdLibrary folder on GitHub.

- MyDriving.DataStore.Abstractions contains the set of interfaces that the rest of the app uses to work with the data. This project in turn uses either MyDriving.DataStore.Azure (for real data) or MyDriving.DataStore.Mock (for fake data during testing).
- MyDriving.AzureClient is a wrapper for the Azure SDK's `MobileServiceClient` object, which in turn hides the details of making HTTP requests to the back end behind a convenient object model. This `AzureClient` wrapper exists so that we can use the same instance of this object for both the data visualization and field gateway roles of the app.
- Tests/MyDriving.UITests contains testing code for Xamarin Test Cloud that is not included in the final app package.

Provided that you've installed Visual Studio and Xamarin on your development machine, have installed the necessary SDKs (described in the Getting started materials), and have restored the NuGet packages for the solution, you should be able to build the app packages, deploy them to test devices or emulators, and walk through them in the debugger.

> **Note** To build and debug iOS apps from Visual Studio, you'll need a Mac OS X machine on your network. Details about this can be found in the Visual Studio and Xamarin documentation on MSDN. You can also use Xamarin Studio on a Mac to build for iOS and Android, and Xamarin Studio on Windows to build for Windows and Android. Using Visual Studio with a networked Mac, however, is the only way to build all three app packages from the same IDE.

Xamarin components

In a cross-platform app technology like Xamarin, it's essential that you can make full use of platform-specific APIs, which for shared code requires a layer that abstracts the differences behind a common interface. This is what Xamarin *components* or *plug-ins* are for, and they provide a huge productivity boost for developers.

There are many free and paid components available from Xamarin and the community. You can explore them at https://components.xamarin.com/ or https://github.com/xamarin/plugins. You bring them into your projects via NuGet.

These components cover all manner of capabilities, from battery status, barcode scanning, and sensor access, to custom controls, mapping, contacts, UI themes, cloud services, and much more. When you begin a Xamarin project, in fact, one of the first things you should do is browse the component catalogs that we linked to earlier for components that both give you access to basic functionality and those that provide richer or more advanced features that you can build on.

> **Note** The difference between components and plug-ins is not functionally important, but plug-ins generally abstract common capabilities across all supported platforms, whereas components might implement specific features that apply to only a single platform. It's always a good idea to check that a component or plug-in that you're interested in using supports all the platforms you want to target.

If you go into various projects in the MyDriving app solution and expand the **References** node, you'll see all the components that we draw upon. Here are some of the most important ones:

- Connectivity, Device Information, Geolocation, Media, and Settings plug-ins: abstractions for common platform APIs.

- HockeySDK.Xamarin (iOS and Android) and HockeySDK.UWP (Windows): the client-side libraries for logging telemetry to HockeyApp. We'll talk more about HockeyApp in Chapter 5.
- Azure Devices Client PCL (Microsoft.Azure.Devices.Client): the client-side Azure library for working with IoT Hub over HTTP. The main object we use from this library is called `DeviceClient`, which we'll encounter later.
- Azure Mobile Client SDK (Microsoft.WindowsAzure.Mobile): the client-side Azure library for working with App Service features, the main object of which is called `MobileServices-Client`. This library provides an object model for working with back end storage tables, method wrappers for calling back-end API endpoints, and helpers for authentication, among other things. It transparently handles all HTTP communication with the back end.
- Acr.UserDialogs: a message box UI for Xamarin apps.

App flows: authentication, IoT field gateway, and data visualization

The MyDriving app plays two distinct roles—that of an IoT field gateway and that of a consumer app. These roles can be served by different apps, and the field gateway role can be fulfilled by separate hardware altogether. In MyDriving, both roles are contained in the same app, resulting in a little overlap between the two. For this reason, it helps to tease apart the code that's related to each role.

For these walkthroughs, assume that a driver has plugged an OBD dongle into their car and paired it with their phone. They then install the MyDriving app and launch it for the first time, and it goes through the process of authentication with the back end that's needed in both roles (that is, authentication via Azure App Services).

After it's authenticated, the app doesn't start acting like an IoT field gateway (collecting data) until the user starts a trip through the app's UI. We point this out because this is the one piece of UI that's related to the IoT role; everything else is related to consuming the *results* from recordings, including results from back-end processing. In other words, this intermix of UI works for the MyDriving scenario, but clearly wouldn't exist in scenarios with a separate app or dedicated device on the IoT side.

For convenience, we'll look primarily at the code for the Android app, so assume we're referring to the MyDriving.Android project unless otherwise noted. The iOS and Windows apps are similar in structure and you should be able to identify the appropriate parts of those projects, especially by searching on the class or method names that are shared across the platforms. A few details about app startup for iOS and Windows are also given in the *iOS Startup* and *Windows Startup* sections later on.

When referring to or showing code, we'll also note the project name and code file, but won't indicate project folders like DataStores because it should be obvious where they live in the solution. For simplicity, we'll also refer to the MyDriving (Portable) project as the MyDriving PCL or just MyDriving when used in path names.

App flow: authentication

Authentication with the back end is needed for both IoT gateway and data visualization roles in the MyDriving app because we need to know which data is associated with which individual user. When both roles are served by the same app, we need do this authentication only once. In scenarios with a different app or device serving as an IoT gateway, the two separate apps or devices take similar steps to authenticate independently.

User identity is handled through a small number of providers such as Facebook, Twitter, and Microsoft (Google is another option, but it isn't implemented in the app's UI). These, plus Azure Active Directory, are the providers that Azure App Service supports, meaning that a driver can use credentials from any of these providers to establish their identity within the whole system.

To allow these connections in your own deployment, you must first visit the developer portals for each of the identity providers you want to support. Then you register your "application," which refers to your whole system of backend plus apps. The registration is what tells the provider to accept authentication requests from that particular application. For details, start with How to configure your App Service application to use Facebook login, and click on the tabs across the top of this page for other providers. (Be sure to always use an https address for the redirect URI.)

The result of a successful authentication is a unique user ID that's stored in the back end's App Service. The client app also gets this user ID along with an authentication token for use with subsequent calls to the App Service API. Let's now see how it all works in the MyDriving app, following the flowchart in Figure 3-4. Note that on its first run, the app shows a Getting Started experience, which we're skipping because it's not essential to the overall app flow.

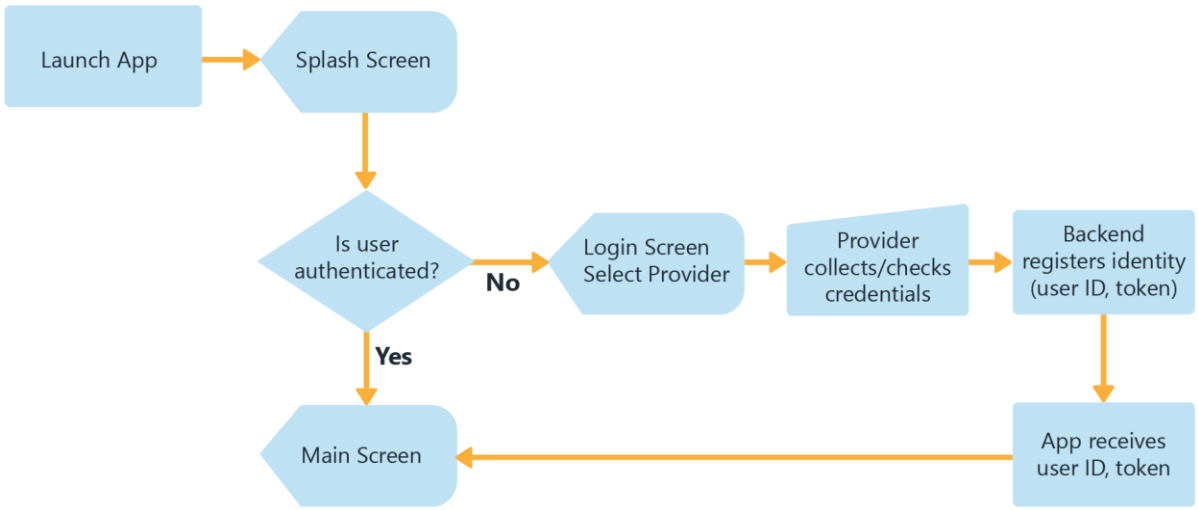


Figure 3-4: The startup flow of the MyDriving app (omitting the Getting Started UI experience)

In the Android app, the entry point where we first run code is `MainApplication.OnCreate` in MainApplication.cs. (Similar entry points are in Main.cs in the iOS project and App.xaml.cs in the Windows project; for more information, see the sections at the end of this chapter.) Here we first see the registration of two classes with a helper class called `ServiceLocator`, in which the `Add` method creates an instance of the class in question and associates it with the given interface type. (A third logger class is omitted.):

```
ServiceLocator.Instance.Add<IAuthentication, Authentication>();
ServiceLocator.Instance.Add<IOBDDevice, OBDDevice>();
```

`ServiceLocator` (MyDriving.Utils/ServiceLocator.cs) is an implementation of the service locator pattern and is the means by which how the platform-neutral shared code in the MyDriving PCL obtains platform-specific implementations of various interfaces at run time. The Android implementation of the `Authentication` class, for example, in MyDriving.Android/Helpers/Authentication.cs. `OBDDevice`, comes from MyDriving.Shared in OBDDevice.cs.

With this bit of wiring in place, the app continues its startup sequence in `SplashActivity.OnCreate` (Activities/SplashActivity.cs):

```
Intent newIntent;

if (Settings.Current.IsLoggedIn)
{
    newIntent = new Intent(this, typeof (MainActivity));
    MyDriving.Services.OBDDataProcessor.GetProcessor().Initialize(
        ViewModel.ViewModelBase.StoreManager);
}
else if (Settings.Current.FirstRun)
{
    newIntent = new Intent(this, typeof (GettingStartedActivity));
    Settings.Current.FirstRun = false;
}
else
    newIntent = new Intent(this, typeof (LoginActivity));

newIntent.AddFlags(ActivityFlags.ClearTop);
newIntent.AddFlags(ActivityFlags.SingleTop);
StartActivity(newIntent);
Finish();
```

As you can see, the app launches `GettingStartedActivity` on first run only; after that it always goes to `LoginActivity` until the user is authenticated, which is the path that concerns us here.

The login screen shown in Figure 3-5 (Activities/LoginActivity.cs and Resources/layout/activity_-login.axml) lets the user choose between Facebook, Microsoft, and Twitter identity providers.*

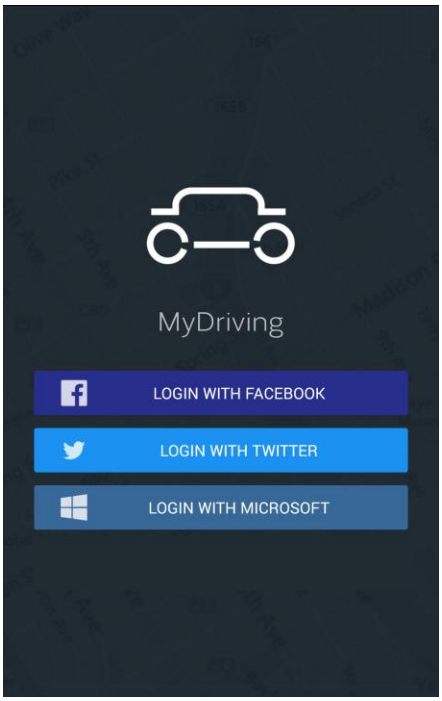


Figure 3-5: The MyDriving app login screen

After you select any one of the identity providers, you end up in this bit of code (LoginActivity.cs):

```
void Login(LoginAccount account)
{
    switch (account)
    {
```

* It should be noted that the MyDriving app doesn't currently have a sign-out feature. If you want to change identities, you need to uninstall and reinstall the app.

```
            case LoginAccount.Facebook:
                viewModel.LoginFacebookCommand.Execute(null);
                break;

            case LoginAccount.Microsoft:
                viewModel.LoginMicrosoftCommand.Execute(null);
                break;

            case LoginAccount.Twitter:
                viewModel.LoginTwitterCommand.Execute(null);
                break;
        }
    }
```

Here, `viewModel` is a previously-created instance of `LoginViewModel` that's implemented in MyDriving/ViewModel/LoginViewModel.cs. Its constructor also uses `ServiceLocator` to retrieve instances of `IAzureClient` and `IAuthentication`, completing that connection with the platform-specific code:

```
client = ServiceLocator.Instance.Resolve<IAzureClient>()?.Client;
authentication = ServiceLocator.Instance.Resolve<IAuthentication>()
```

We know that the `IAuthentication` instance comes from the Android-specific code we saw earlier. The `IAzureClient` is an instance of the `AzureClient` object (MyDriving.AzureClient/AzureClient.cs), which is again a wrapper around the Azure SDK's `MobileServiceClient` object and is how we talk to the back end's App Service component located at https://mydriving.azurewebsites.net. `AzureClient` is registered with `ServiceLocator` in the `ViewModelBase` class (MyDriving/ViewModel/ViewModel-Base.cs) from which `LoginViewModel` derives.

Each of the view model's `Execute` methods call `LoginViewModel.LoginAsync` with the chosen provider:

```
async Task<bool> LoginAsync(MobileServiceAuthenticationProvider provider)
{
    // [Connectivity check and other error handling omitted]

    MobileServiceUser user = null;

    authentication.ClearCookies();
    user = await authentication.LoginAsync(client, provider);

    // [...]
}
```

The call to `Authentication.LoginAsync` (MyDriving.Android/Helpers/Authentication.cs) in turn calls `AzureClient.LoginAsync`:

```
public async Task<MobileServiceUser> LoginAsync(IMobileServiceClient client,|
    MobileServiceAuthenticationProvider provider)
{
    var user = await client.LoginAsync(CrossCurrentActivity.Current.Activity, provider);
    Settings.Current.AuthToken = user?.MobileServiceAuthenticationToken ?? string.Empty;
    Settings.Current.AzureMobileUserId = user?.UserId ?? string.Empty;
    return user;
}
```

`client.LoginAsync` is a pass-through to the Azure SDK's `MobileServicesClient.LoginAsync` method, which displays UI within the current screen to collect credentials (Figure 3-6). The provider then validates those credentials, prompts the user to authorize the application (Figure 3-7), and sends the resulting user ID to App Service, which saves it. This user ID, along with the authentication token for the client, is then returned as a `MobileServiceUser` object, as you can see in the previous code.

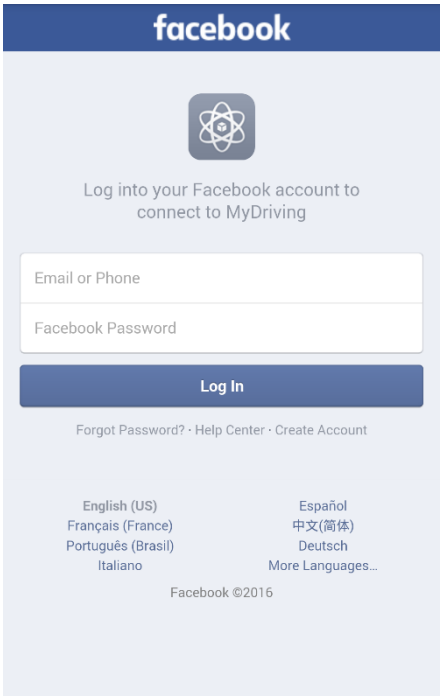


Figure 3-6: Initial login screen for Facebook

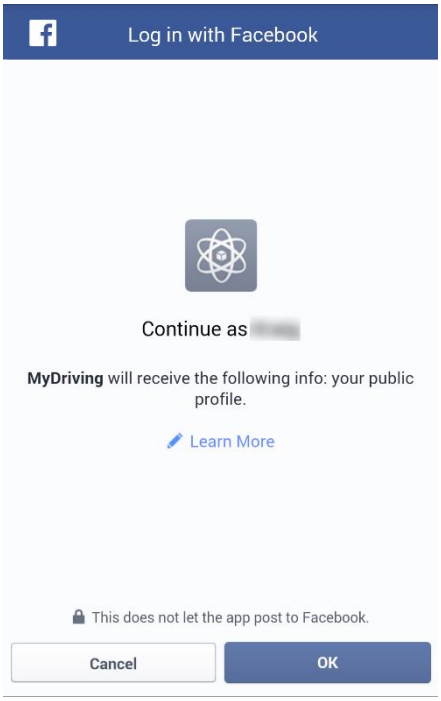


Figure 3-7: Facebook authorization screen

After a user is authenticated, the app saves a `UserProfile` object to the back end, which can be accessed through either the UserProfile table or the */api/userinfo* endpoint. The difference is that the latter will retrieve the latest details that come from the underlying identity provider (such as the user's latest profile photograph), while the former has information that can get stale.

When a user runs the app at a later time, we're able to skip the login screen. Going back to `LoginViewModel.LoginAsync` (MyDriving/ViewModels/LoginViewModel.cs), there's one bit we didn't show earlier:

```
authentication.ClearCookies();
user = await authentication.LoginAsync(client, provider);

if (user != null)
{
    IsBusy = true;
    UserProfile = await UserProfileHelper.GetUserProfileAsync(client);
}
```

The `UserProfileHelper.GetUserProfileAsync` method (MyDriving/Helpers/UserProfile.cs) is what invokes the UserInfo API to get updated details:

```
public static async Task<UserProfile> GetUserProfileAsync(IMobileServiceClient client)
{
    var userprof = await client.InvokeApiAsync<UserProfile>(
        "UserInfo",
        System.Net.Http.HttpMethod.Get,
        null);

    // [...]

    return userprof;
}
```

App flow: IoT field gateway

Now that the user is authenticated, we can start having some fun with data! At this point, the app shows the Current Trip main screen (Figure 3-8), which is implemented as a fragment inside the Main screen. (Main is in Activities/MainActivity.cs and Resources/layout/activity_main.axml; past trips are in Fragments/FragmentCurrentTrip.cs and Resources/layout/fragment_current_trip.axml.) The Current Trip UI is the one that concerns us here in the app's role as a field gateway. The prominent record button is what starts the flow of data.

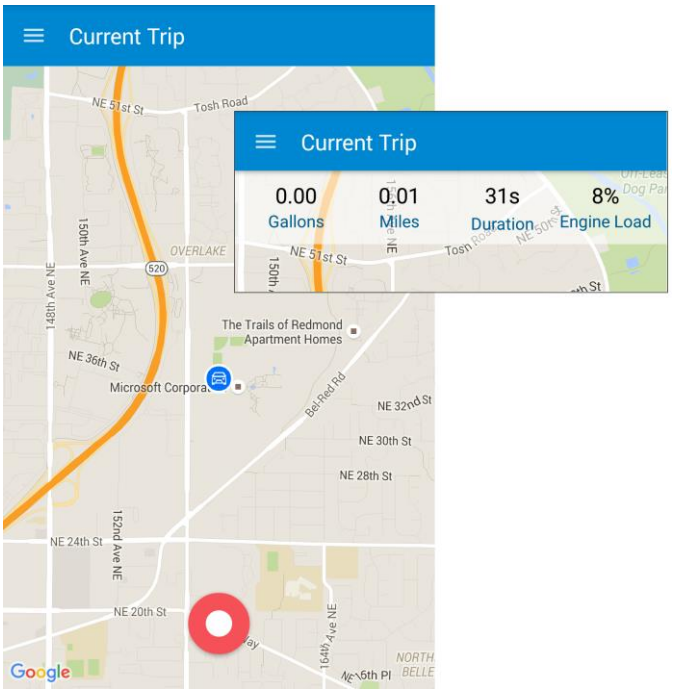


Figure 3-8: The current trip UI before recording. The inset shows overlay after the recording has started.

Before we tap the record button, however, we must jump all the way back for a moment to the app startup sequence in `SplashActivity.OnCreate` (Activities/SplashActivity.cs), where we see this very important line:

```
// When the first screen of the app is launched after user has logged in, initialize
// the processor that manages connection to the OBD device and to IoT Hub.
MyDriving.Services.OBDDataProcessor.GetProcessor().Initialize(
    ViewModel.ViewModelBase.StoreManager);
```

The `OBDDataProcessor` class, in the MyDriving/Services/OBDDataProcessor.cs, is the bridge to the OBD device library. There's one static instance in the app that's returned by its `GetProcessor` method:

```
public class OBDDataProcessor
{
    static OBDDataProcessor _obdDataProcessor;

    // [...]

    public static OBDDataProcessor GetProcessor()
    {
        return _obdDataProcessor ?? (_obdDataProcessor = new OBDDataProcessor());
    }

    // [...]

}
```

Now we get to the all-important `Initialize` method, where we see the use of `ServiceLocator` to retrieve the `IOBDDevice` implementation that was registered at startup:

```
//[This line is in the OBDDataProcessor constructor]
IHubIOT iotHub = new IOTHub();

public async Task Initialize(IStoreManager storeManager)
{
    //Ensure that initialization is only performed once
    if (!_isInitialized)
    {
        isInitialized = true;
        this.storeManager = storeManager;

        //Get platform-specific implementation of IOBDDevice
        obdDevice = ServiceLocator.Instance.Resolve<IOBDDevice>();

        //Start listening for connectivity change event so that we know if
        //connection is reestablished\dropped when pushing data to the IoT hub
        CrossConnectivity.Current.ConnectivityChanged +=
            Current_ConnectivityChanged;

        //Provision the device with the IoT hub
        var connectionStr = await
            DeviceProvisionHandler.GetHandler().ProvisionDevice();
        iotHub.Initialize(connectionStr);

        //Check right away if there is any trip data left in the buffer that
        //needs to be sent to the IoT hub - run this thread in the background
        SendBufferedDataToIOTHub();
    }
}
```

The two most important steps here are "provisioning" the device and initializing the connection to IoT Hub, which are bound to each other through a connection string.

Provisioning the device means that we give the back end a unique device ID along with any other information we want to associate with it, such as a user ID. The app sends this information using the back end's */api/provision* endpoint that's discussed in Chapter 4. You can see the exact mechanics in the `AzureClient.ProvisionDevice` method in MyDriving.AzureClient/DeviceProvisionHandler.cs:

```
public async Task<string> ProvisionDevice()
{
    // [Some error checking omitted]

    Dictionary<string, string> myParms = new Dictionary<string, string>();
    myParms.Add("userId", Settings.Current.UserUID);
    myParms.Add("deviceName", Settings.Current.DeviceId);

    var client = ServiceLocator.Instance.Resolve<IAzureClient>()?.Client
        as MobileServiceClient;

    try
    {
        var response = await client.InvokeApiAsync("provision", null,
            HttpMethod.Post, myParms);
        AccessKey = response.Value<string>();
    }
    catch { /* ... */ }

    // The DeviceConnectionString is created with the AccessKey (see below)
    Settings.Current.DeviceConnectionString = DeviceConnectionString;
    return Settings.Current.DeviceConnectionString;
}

public string DeviceConnectionString
{
    get
    {
```

```
            string connectionStr = String.Empty;
            if (!String.IsNullOrEmpty(AccessKey) && !String.IsNullOrEmpty(HostName)
                && !String.IsNullOrEmpty(DeviceId))
            {
                // HostName is set to "mydriving.azure-devices.net" by default
                connectionStr =
                    $"HostName={HostName};DeviceId={DeviceId};SharedAccessKey={AccessKey}";
            }

            return connectionStr;
        }
    }
```

Within the API code, the back end makes sure it can handle another device (that is, it hasn't reached its limit), saves the device ID with the associated data, and hands back an access key that must be included in the connection string for the endpoint to which we send data. In short, provisioning means asking the IoT hub for permission to send data to it, and the access key is how it says "yes."

With an authorized connection string in hand, we can now plug ourselves—which is to say, the MyDriving app playing the field gateway role—into the IoT hub for real. The `iotHub.Initialize` method (MyDriving/Services/IOTHub.cs) internally creates an Azure IoT Hub `DeviceClient` object from the connection string, whose `SendEventAsync` method is what we use to upload data. Here's the whole `IOTHub` helper class. (Null checks on function arguments are omitted for brevity.):

```
public class IOTHub : IHubIOT
{
    private DeviceClient deviceClient;

    public void Initialize(string connectionStr)
    {
        deviceClient = DeviceClient.CreateFromConnectionString(connectionStr,
            TransportType.Http1);
    }

    public async Task SendEvents(IEnumerable<String> blobs)
    {
        List<Message> messages = blobs.Select(
            b => new Message(Encoding.ASCII.GetBytes(b))).ToList();
        await deviceClient.SendEventBatchAsync(messages);
    }

    public async Task SendEvent(string blob)
    {
        var message = new Message(Encoding.UTF8.GetBytes(blob));
        await deviceClient.SendEventAsync(message);
    }
}
```

Note Because we're using the PCL version of the Azure IoT Devices SDK, the `DeviceClient.CreateFromConnectionString` method allows a choice of protocol. This is a departure from existing documentation on the `DeviceClient` object that indicates use of the AMQPS protocol.

With this connection, the phone is now set up as a field gateway, and we're ready to collect and transmit data when the user taps the record button. Doing so starts the following flow (omitting UI-related steps):

1. The app creates a new instance of the `Trip` class that manages the recording as a series of `TripPoint` objects. (See MyDriving.DataObjects/Trip.cs and TripPoint.cs for these classes.)
2. The app connects to the OBD device and starts polling for data at one-second intervals, keeping the latest reading in memory. (For demonstration purposes, the app uses simulated data if there isn't an OBD device at all.)

3. The app listens for position-changed events from the phone's GPS sensor, which are configured to fire at three-second or 5-meter intervals.

4. For each position-changed event, the app creates a new `TripPoint` with the current geolocation and OBD data and sends it to the IoT hub through the `IOTHub.SendEvent` method seen earlier. (In the UI, a few current trips stats are also shown.)*

5. The listeners remain active until the user stops recording.

> **Note** The use of geolocation events as the trigger for reading OBD data and uploading to the IoT hub depends heavily on having a reliable geolocator in the device. In your own scenarios, be sure that the trigger you're using meets your streaming requirements for IoT data.

Let's follow this flow in the Android code. First, the Current Trip screen creates a `CurrentTripViewModel` (MyDriving/ViewModel/CurrentTripViewModel.cs), to manage the `Trip` instance (along with the unused *Photo* list, MyDriving.DataObjects/Photo.cs) and a copy of the `OBDDataProcessor`:

```
public class CurrentTripViewModel : ViewModelBase
{
    readonly OBDDataProcessor obdDataProcessor;
    readonly List<Photo> photos;
    public IGeolocator Geolocator => CrossGeolocator.Current;
    // [...]

    public CurrentTripViewModel()
    {
        CurrentTrip = new Trip
        {
            UserId = Settings.Current.UserUID,
            Points = new ObservableRangeCollection<TripPoint>()
        };
        photos = new List<Photo>();

        // [Initialize other fields for UI data binding; omitted]

        obdDataProcessor = OBDDataProcessor.GetProcessor();
    }
    // [...]
}
```

That completes step 1 of our flow. Next, tapping the record button goes to `OnRecordButtonClick` (Fragments/FragmentCurrentTrip.cs), which calls `CurrentTripViewModel.StartRecordingTrip`:

```
await viewModel.StartRecordingTrip();
```

Within `StartRecordingTrip`, we connect to the OBD device and record the first trip point:

```
await obdDataProcessor.ConnectToObdDevice(true);
CurrentTrip.RecordedTimeStamp = DateTime.UtcNow;

CurrentTrip.Points.Add(new TripPoint
{
    RecordedTimeStamp = DateTime.UtcNow,
    Latitude = CurrentPosition.Latitude,
    Longitude = CurrentPosition.Longitude,
    Sequence = CurrentTrip.Points.Count,
});
```

The `ConnectToObdDevice` method (MyDriving/Services/OBDDataProcessor.cs) gets the data stream started. It calls into `IOBDDevice.Initialize` (MyDriving.Shared/ObdDevice.cs), which wends its way to `ObdWrapper.Init` in Utilities/ObdLibAndroid/ObdWrapper.cs. That's where you'll find a few hundred

* The app project has code for caching photos during a trip, which are saved to cloud storage when recording stops. This shows how a field gateway app could add its own features to a scenario. MyDriving, however, does not enable taking photos in the UI.

lines of code that connects to the OBD dongle via a Bluetooth socket and then enters a continuous loop inside the `PollObd` method. This method requests values for all the OBD PIDs defined by the `ObdUtil.GetPIDs` method (Utilities/ObdShare/ObdUtil.cs, as also listed in Chapter 10, *Reference,* under *OBD data schema*). This polling maintains the most recent values in a dictionary instance called `data`, which are then returned whenever `OBDWrapper.Read` is called. This completes step 2 of the flow.

For step 3, the geolocation listener is set up inside `CurrentTripViewModel.ExecuteStartTracking-TripCommandAsync` (a method that eventually gets called when the GPS is initialized; this code is in CurrentTripViewModel.cs):

```
Geolocator.PositionChanged += Geolocator_PositionChanged;
await Geolocator.StartListeningAsync(3000, 5);
```

Every three seconds or every 5 meters, whichever comes first, we'll get a call to `Geolocator_-PositionChanged`, where we then fulfill step 4 (error checking and other details omitted):

```
async void Geolocator_PositionChanged(object sender, PositionEventArgs e)
{
    if (IsRecording)
    {
        var userLocation = e.Position;
        // [Check for spurious GPS readings (omitted)]

        var point = new TripPoint
        {
            TripId = CurrentTrip.Id,
            RecordedTimeStamp = DateTime.UtcNow,
            Latitude = userLocation.Latitude,
            Longitude = userLocation.Longitude,
            Sequence = CurrentTrip.Points.Count,

            // [Set default for OBD values]
        };

        // Add OBD data
        await AddOBDDataToPoint(point);

        // Add the point to the trip
        CurrentTrip.Points.Add(point);

        // Push the trip data packaged with the OBD data to the IoT hub
        obdDataProcessor.SendTripPointToIOTHub(CurrentTrip.Id, CurrentTrip.UserId, point);

        // [Do a bunch of calculations for cumulative values like distance,
        // gas usage, and driving time]
    }

    CurrentPosition = e.Position;
}
```

The `AddOBDDataToPoint` method, as you probably guessed, calls `OBDDataProcessor.ReadOBDData`, which calls `OBDDevice.ReadData`, which then goes to `OBDWrapper.Read`, which—breathe!—returns the most recently polled OBD data values as we saw earlier.

And it's no mystery that `OBDDataProcessor.SendTripPointToIOTHub` basically packages up the current `tripId`, the current `userId`, and the `TripPoint` into a blob of JSON and shoots it off to the back end through the `IOTHub.SendEvent` method that we saw earlier. The blob basically looks like this:

```
{
  {
     "TripId" : <tripID>,
     "UserId" : <userID>
  },
  "TripDataPoint" : {
    <serialized TripPoint object>
  }
}
```

Eventually, of course, the trip comes to an end and recording stops (step 5). This brings us into `CurrentViewModel.StopRecordingTrip` and `SaveRecordingTripAsync` for some wrap-up tasks:

- Turn off the OBD connection, which disconnects the Bluetooth socket (follow through from `OBDDataProcessor.DisconnectFromObdDevice` to `OBDWrapper.Disconnect`).
- Prompt the user for a name to assign to the trip, which eventually appears in the Past Trips UI as we'll see shortly.
- Grab a representative picture for the recording from dev.virtualearth.net based on the ending position of the trip.
- Finalize the `Trip` object with all its `TripPoint` objects and save it with a timestamp in the back end's App Service storage tables. This storage is managed through the code in the MyDriving.DataStore.Abstractions project. In production, this project ultimately talks again to the Azure `MobileServicesClient` object to sync the data with the cloud.

Notice that we're saving information only to storage here—nothing more is being sent to the IoT hub. That's because the back end's processing system is interested in only real-time IoT data. The rest is of interest only to the app.

> This demonstrates a best practice for mobile apps: stream what data the back end needs for processing, but keep the rest in locally-cached and synchronized cloud storage such as what App Service provides. The reason for this is that an app can't rely on a constant Internet connection. By using App Service tables, with their available sync features, the app doesn't really have to concern itself with connectivity. Instead, the app can just write to storage in its IoT field gateway role and then turn around and reliably read from that storage in its data visualization role, even if there's no connection. App Service automatically keeps the tables in sync when connectivity is restored.
>
> Furthermore, because the back end also writes its results to those same tables, the app (in its consumption role) doesn't need to separately request those results. When it retrieves data from the same table where it stored the trip originally, it will find the results from the back end inserted there. (The back end's processing has usually completed by the time you navigate the UI.)
>
> If we didn't use this pattern and instead tried to stream *everything* to the back end—caching nothing locally—then it might be possible to record a trip but not to view it right away. That's because, for example, the phone could lose connectivity between recording and viewing. With the design we're using here, we can see everything immediately in the app even if connectivity is lost, except for the few results from the back end.

So what *has* the back end been doing with the streaming IoT data all this time? Besides saving data for use by Power BI and for retrieval from the app, it's been happily applying Stream Analytics and Machine Learning to see what patterns it can derive from it. In particular, it's been looking for hard accelerations and hard stops, and saving those "points of interest" into the App Service table called `POI`. It's also using that information to produce a driving style assessment that's saved in the `Trip.Rating` property.

In fact, some of that information is already available when we stop recording a trip. Remember that the app, even in the IoT field gateway role, is already authenticated with that App Service. Thus it can read all the tables in App Service where the back end has been storing its results. It can do this even within the `StopRecordingTrip` method, where it retrieves the points of interest to save with the trip:

```
List<POI> poiList = new List<POI>();
poiList = (List<POI>)await StoreManager.POIStore.GetItemsAsync(CurrentTrip.Id);
CurrentTrip.HardStops = poiList.Where(p => p.POIType == POIType.HardBrake).Count();
CurrentTrip.HardAccelerations = poiList.Where(p => p.POIType ==
    POIType.HardAcceleration).Count();
```

All of this data, then, will be immediately available for visualization as we'll see in the next section. But first we must mention that some of this data, along with a few other aggregate values from the trip,

are also saved in a `TripSummaryViewModel` instance. This data is used by the summary screen (Figure 3-9) that appears when recording is complete (generated from Activities/TripSummaryActivity.cs).

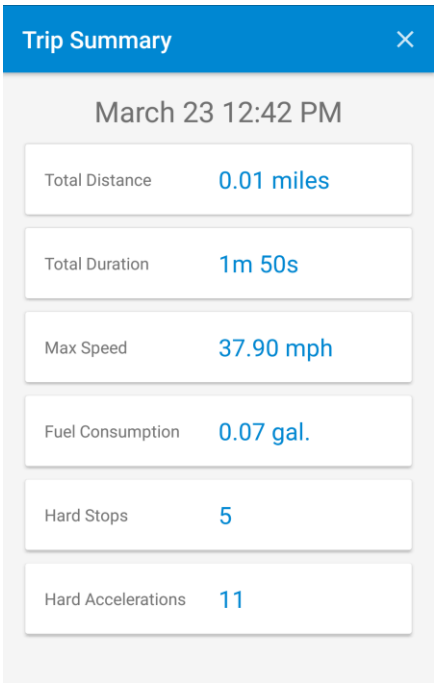


Figure 3-9: The trip summary UI after recording

App flow: data visualization

In the previous section, we saw how a user records a trip, an activity that sends a bunch of IoT data to the back end via the IoT hub while the trip is happening, and then stores route information with pictures into an App Service storage table when the trip is complete. Whew! It was a lot of work, but you should now have a pretty clear idea of how a field gateway really operates, whether it's implemented as part of an app, as in MyDriving, or implemented in a dedicated device.

We'll switch gears now to the app's data visualization and consumption role, which of course does not have to be a mobile app experience at all. Because all trip data is available through REST APIs, you can create any number of visualization experiences around those same results that have nothing to do with mobile devices. But again, because most drivers already have a mobile phone, it makes sense in the MyDriving scenario to use the phone for the consumer experience too.

That experience, as we saw in Chapter 1, *Introduction*, is centered on viewing past trips in an interesting and engaging manner that integrates the results coming from the back end. In the MyDriving app, this happens in two ways:

- Viewing past trips that include a driver rating and points of interest (hard stops or accelerations) for that trip. This happens when you tap on a past trip that appears on the main screen. The user can return to the main screen through the **Past Trips** command on the menu.
- Viewing an aggregate driver rating for all past trips combined, which happens when the user taps the **Profile** command on the app menu.

Remember that all recorded trips are always available through the `Trip` table in the App Service storage. That's because we saved each trip there when recording was finished. And because the machine learning module automatically stores the trip's points of interest in the `POI` table, we already have everything we need to display those past trips in the UI.

To view past trips, open the app's menu and select **Past Trips**. This loads the past trips fragment (Fragments/FragmentPastTrips.cs and Resources/layout/fragment_past_trips.axml) into the Main screen UI as shown in Figure 3-10.

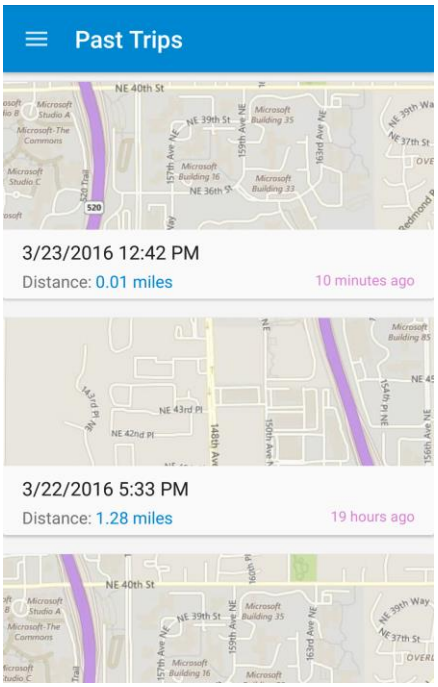


Figure 3-10: The main screen showing past trips

When this view comes up, we enter into its `OnCreateView` method (Fragments/FragmentPastTrips.cs):

```
public override View OnCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState)
{
    // [...]

    viewModel = new PastTripsViewModel();
    refresher = view.FindViewById<SwipeRefreshLayout>(Resource.Id.refresher);
    refresher.Refresh += (sender, e) => viewModel.LoadPastTripsCommand.Execute(null);

    // [...]

    return view;
}
```

The view clearly gets its data from an instance of the `PastTripsViewModel` class in the MyDriving PCL (MyDriving/ViewModel/PastTripsViewModel.cs), which is instructed here to load past trips right away. The `LoadPastTripsCommand.Execute` call lands us in the view model's `ExecuteLoadPastTrips-CommandAsync` method, where the following line is the crux of the scene:

```
Trips.ReplaceRange(await StoreManager.TripStore.GetItemsAsync(0, 25, true));
```

`Trips` is just an `ObservableCollection<Trips>`, and so here we're asking the `StoreManager` object to get the data for us (25 items at a time).

We encountered `StoreManager` earlier with the IoT field gateway part of the app, but we didn't go into any details. Let's do that now, because it's the mechanism through which the app talks to the back-end storage in App Service.

`StoreManager` (which ultimately comes from MyDriving.DataStore.Abstractions/IStoreManager.cs by way of the service locator) is an abstraction layer for the storage tables that allows us to hide a number of details from the rest of the app, including:

- Swapping in mock data for use during testing.
- Using the Azure SDK `MobileServiceClient` for table access (which in turn hides the details of HTTP interactions).
- Using the `MobileServiceClient` object's "sync table" features to ensure that the app always has the most recent data. (See Enable offline sync for your mobile app for more information about sync tables.)

The actual mechanics of all this aren't important for our discussion here, so we'll leave you to look through that code on your own. The important thing to understand is that a member like `TripStore` in the `StoreManager` is how we work with the underlying `Trip` table. There is, in fact, one `*Store` class for each table in the back end. You'll find these in the MyDriving.DataStore.Azure project in the Stores folder. (The mocks are in MyDriving.DataStore.Mock.) Each one in turn derives from `BaseStore<T>` in BaseStore.cs, which does the mapping to the underlying sync table:

```
protected IMobileServiceSyncTable<T> Table => table ?? (table =
    ServiceLocator.Instance.Resolve<IAzureClient>()?.Client?.GetSyncTable<T>());
```

TripStore.cs, then, is where we find the implementation of `TripStore.GetItemsAsync`, as called when the past trips UI is coming up. As you'd expect, it simply retrieves the objects from storage into memory (`Photos`, again, is not used but exists in the code):

```
public override async Task<IEnumerable<Trip>> GetItemsAsync(int skip = 0,
    int take = 100, bool forceRefresh = false)
{
    var items = await Table.Skip(skip).Take(take).OrderByDescending(
        s => s.RecordedTimeStamp).ToEnumerableAsync().ConfigureAwait(false);

    foreach (var item in items)
    {
        item.Photos = new List<Photo>();
        var photos = await photoStore.GetTripPhotos(item.Id).ConfigureAwait(false);
        foreach (var photo in photos)
            item.Photos.Add(photo);
    }

    return items.OrderByDescending(s => s.RecordedTimeStamp);
}
```

Again, what's in that storage is both the data that the app wrote there when we finished recording a trip, and the results that the back end writes there in the course of processing the IoT data stream.

What, then, do we have for each `Trip` object? You can see that in the class definition of MyDriving.DataObjects/Trips.cs (also spelled out in Chapter 10, in the section "App Service tables"). Most of the properties were what the app stored once a recording was completed, including some calculated values like total trip distance and total time.* But a few—namely, `Rating`, `HardStops`, and `HardAccelerations`—were inserted by the back end as it ran all the IoT data through services like Stream Analytics and Machine Learning, as we'll see in later chapters.

The bottom line, though, is that when we're focusing on data visualization here, we don't have to worry about any of that. We simply trust that we can retrieve what's now a *complete* `Trip` record and present it meaningfully to the user.

* In this particular scenario, such calculations weren't particularly demanding and were easy to do in the mobile app acting as a field gateway. Other calculations, though, might be better left to the backend. When designing your own system, you'll want to look at every such computed property and decide where best to do the work.

The main screen's list of Past Trips displays just a few parts of each `Trip`. Tapping on an individual trip then takes the user to the details screen (Activities/PastTripDetailsActivity.cs and Resources/layout/-activity_past_trip_details.axml). As you can guess by now, this is a view for `PastTripsDetailViewModel` (MyDriving/ViewModel/PastTripsDetailViewModel.cs) that manages a single `Trip` object. The view (Figure 3-11) simply shows all the details in an interactive display that lets you move through the points along the route to see accumulated telemetry and points of interest.

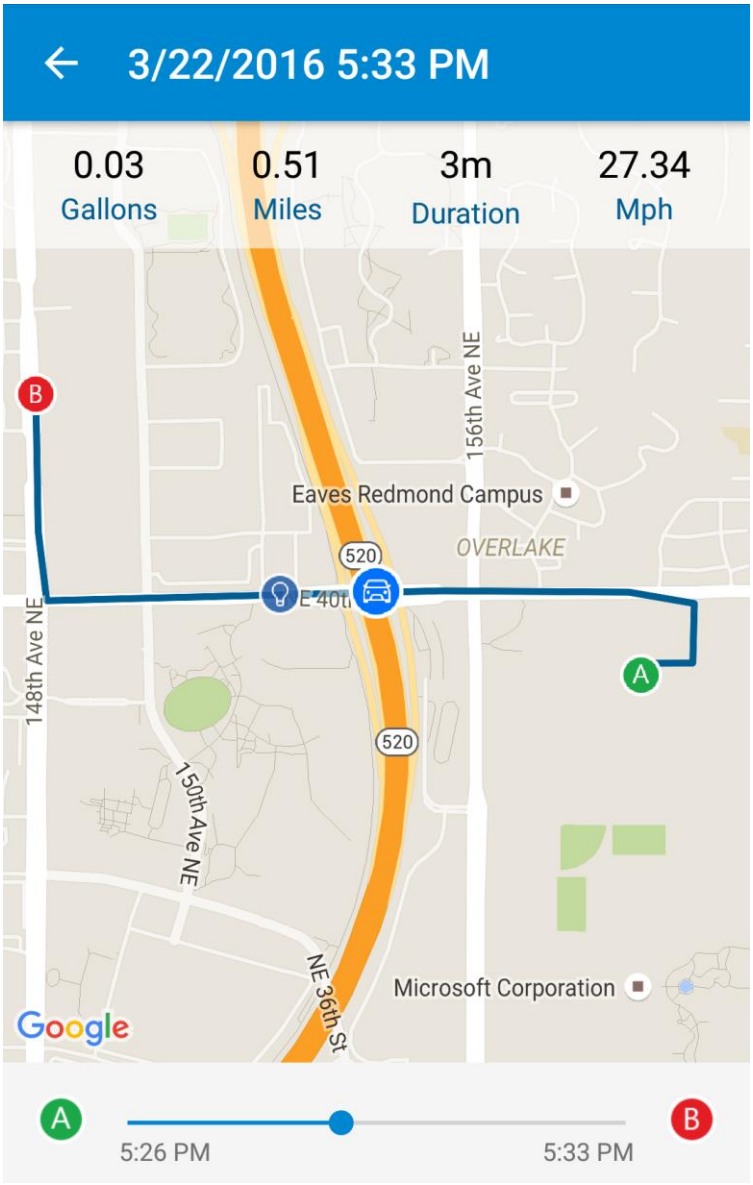


Figure 3-11: The past trip details screen

Now for the *coup de grace*. The most important piece of human value that MyDriving delivers, as described in the introduction, is a greater awareness of your overall driving habits. Each past trip shows details like hard stops and accelerations, but the greatest value comes from seeing those numbers, and an overall driver rating, across many trips.

That's what the Profile screen is for (Figure 3-12). It shows a driver's accumulated rating and other statistics like total miles, total time, average speeds (local and highway), and the effective wear-and-tear on the vehicle in terms of hard stops and accelerations.

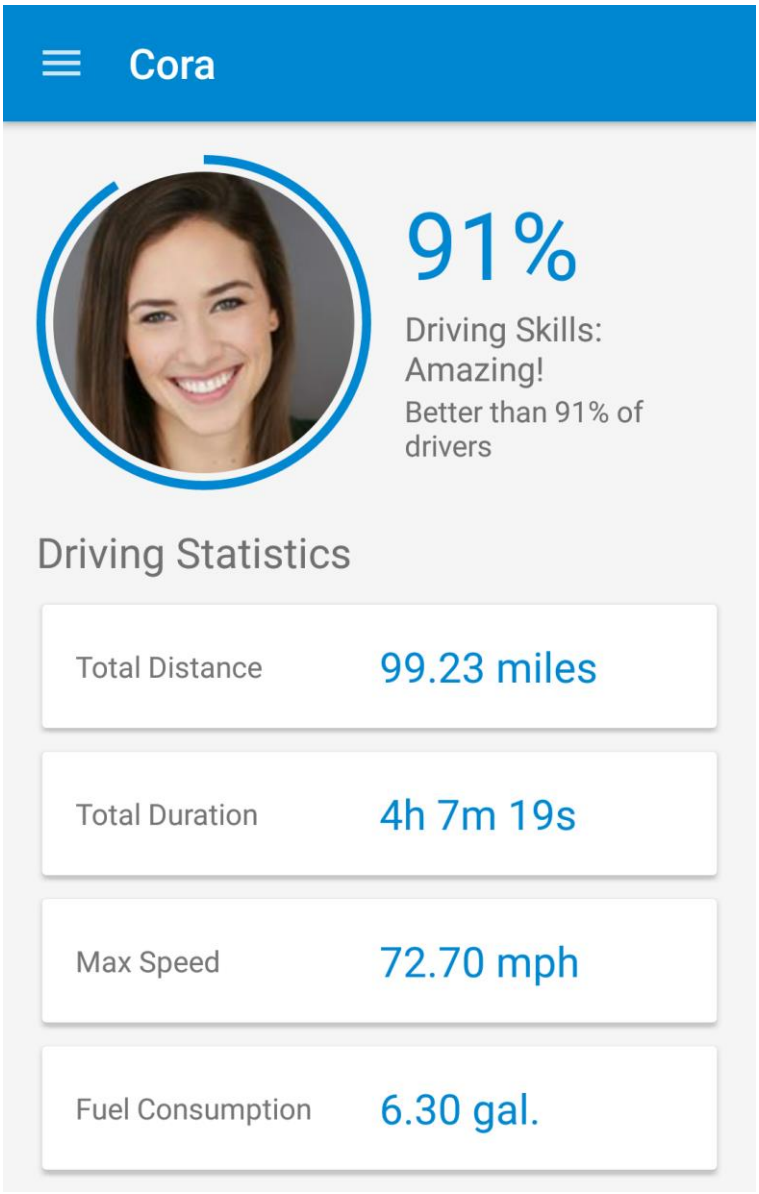


Figure 3-12: The Profile screen.

The Profile screen is another fragment (Fragments/FragmentProfile.cs and Resources/layout/fragment_profile.axml) that's loaded into the context of the Main screen. This view draws from the `ProfileViewModel` class (MyDriving/ViewModel/ProfileViewModel.cs), which simply pulls data from table storage as we've seen elsewhere. In this case, it's `StoreManager.UserStore` that's connected to the underlying UserProfile table:

```
var currentUser = await StoreManager.UserStore.GetItemAsync(Settings.UserUID);
```

Note that the app could also call the */api/userinfo* endpoint in the back end to get an updated profile picture, if desired. The `UpdatePictureAsync` method in this view model does exactly that, but isn't currently used in the app.

In any case, we've completed the primary story of the mobile app in both its roles. The next two sections provide more about the startup sequences of the iOS and Windows apps, but feel free to skip them and go to Chapter 4, where we'll discuss the API endpoints in App Service.

iOS startup (MyDriving.iOS project)

The iOS app code begins with the `Application.Main` method of Main.cs (the standard entry point of the app), which launches the UI by passing the name of the startup *delegate* to `UIApplication.Main`:

```
public class Application
{
    static void Main(string[] args)
    {
        UIApplication.Main(args, "TripApplication", "AppDelegate");
    }
}
```

`AppDelegate` is the main application class that's responsible for creating the `Window` object, building the UI, and listening to events. (For more details see Hello, iOS on Xamarin.com.) A key event is `AppDelegate.FinishedLaunching` (AppDelegate.cs), where we do the same sorts of things as in the Android app: set up the `ServiceLocator` instances, initialize the `OBDProcessor`, and then go to the getting started and login screens if the user isn't authenticated yet (otherwise, we go to the UI view controller that's set in the storyboard; some code omitted here):

```
// ViewModelBase.Init sets up ServiceLocator instances for the data model,
// see MyDriving/ViewModel/ViewModelBase.cs
ViewModel.ViewModelBase.Init();

ServiceLocator.Instance.Add<IAuthentication, Authentication>();
ServiceLocator.Instance.Add<ILogger, PlatformLogger>();
ServiceLocator.Instance.Add<IOBDDevice, OBDDevice>();

if (!Settings.Current.IsLoggedIn)
{
    if (Settings.Current.FirstRun)
    {
        var viewController = UIStoryboard.FromName("Main", null)
            .InstantiateViewController("gettingStartedViewController");
        var navigationController = new UINavigationController(viewController);
        Window.RootViewController = navigationController;
        Settings.Current.FirstRun = false;
    }
    else
    {
        var viewController = UIStoryboard.FromName("Main", null)
            .InstantiateViewController("loginViewController");
        Window.RootViewController = viewController;
    }
}
else
{
    // Window.RootViewController is set in Main.Storyboard

    Services.OBDDataProcessor.GetProcessor().Initialize(
        ViewModel.ViewModelBase.StoreManager);

    var tabBarController = Window.RootViewController as UITabBarController;
    tabBarController.SelectedIndex = 1;
}
```

The login screen, implemented in Screens/LoginViewController.cs, has again a `LoginAsync` method that picks up the button taps. It calls the same methods in the MyDriving PCL's `LoginViewModel` code as the Android app. In other words, the iOS app has a unique view for this screen (and all others), but uses the shared view models in the PCL.

In this context, it's worth showing the app's storyboard (Figure 3-13, next page), especially the ability to open it directly in Visual Studio (which is possible as long as you've completed the necessary iOS installation steps for both your Windows and Mac machines).

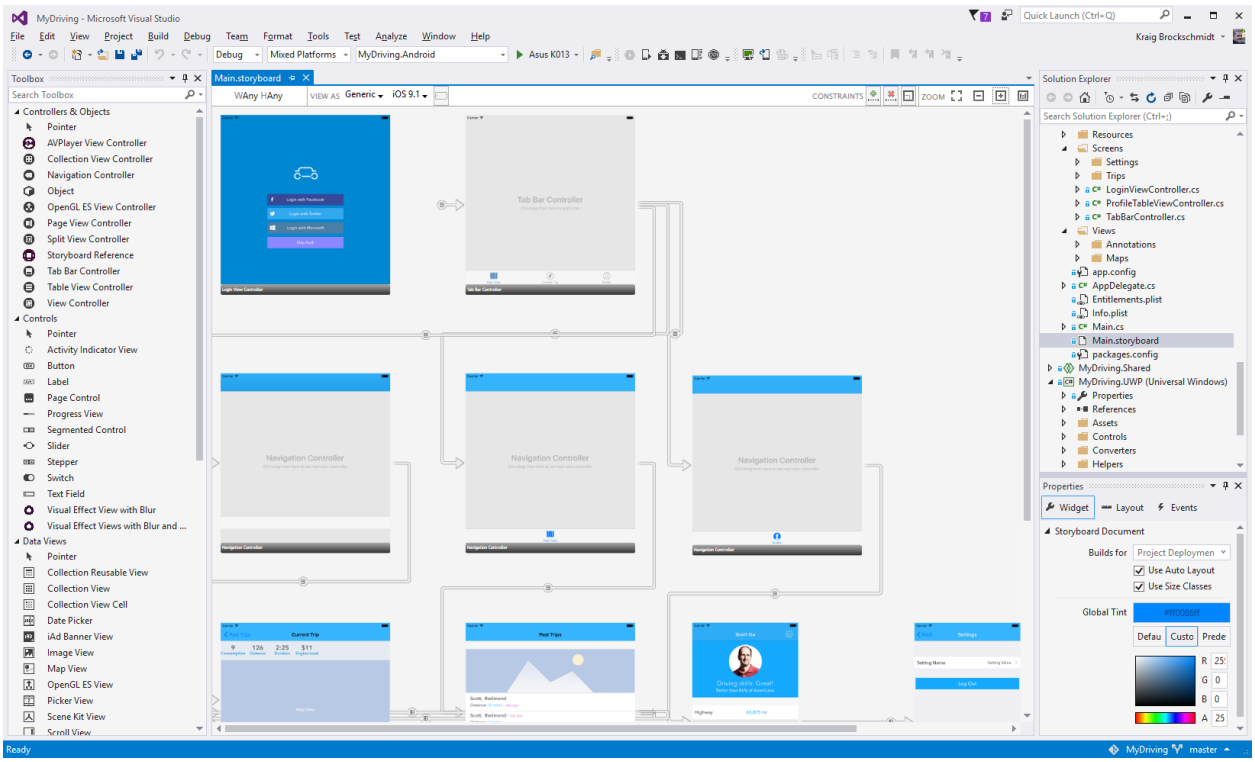


Figure 3-13: The iOS app's Main.storyboard open in Visual Studio

Windows startup (MyDriving.UWP project)

The UWP app code begins in the `App` object constructor and its `OnLaunched` event inside App.xaml.cs. It does the same series of operations as the other platform apps, except that `ViewModelBase.Init()` happens in the constructor (some error checking omitted):

```
public App()
{
    // [...]

    ViewModel.ViewModelBase.Init();

    // [...]
}

protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    SplitViewShell shell = Window.Current.Content as SplitViewShell;

    if (shell == null)
    {
        Frame rootFrame = new Frame();
        rootFrame.NavigationFailed += OnNavigationFailed;
        ServiceLocator.Instance.Add<IAuthentication, Authentication>();
        ServiceLocator.Instance.Add<Utils.Interfaces.ILogger, PlatformLogger>();
        ServiceLocator.Instance.Add<IOBDDevice, OBDDevice>();

        if (Settings.Current.IsLoggedIn)
        {
            Services.OBDDataProcessor.GetProcessor().Initialize(
                ViewModel.ViewModelBase.StoreManager);

            shell = new SplitViewShell(rootFrame);
            shell.SetTitle("CURRENT TRIP");
            shell.SetSelectedPage("CURRENT TRIP");
```

```
                rootFrame.Navigate(typeof(CurrentTripView), e.Arguments);
                Window.Current.Content = shell;
            }
            else if (Settings.Current.FirstRun)
            {
                rootFrame.Navigate(typeof(GetStarted1), e.Arguments);
                Window.Current.Content = rootFrame;
            }
            else
            {
                rootFrame.Navigate(typeof(LoginView), e.Arguments);
                Window.Current.Content = rootFrame;
            }
        }

        Window.Current.Activate();
    }
```

Again, you can see that we go to `LoginView` if the user is not authenticated (Views/LoginView.xaml). In this case, the view doesn't have its own login method; that's because the XAML binds the provider buttons directly to the commands (for example, `LoginTwitterCommand`) in the view model.

App Service API Endpoints

In its dual role of IoT field gateway and data visualization, the mobile app interacts with the back end in two ways: through a provisioning API and IoT Hub when sending trip data, and through API endpoints when retrieving past trips and user profile information. Because the details of IoT Hub are handled by an SDK, that part of the architecture is covered in Chapter 6, *Real-Time Data Handling*, in the section "IoT Hub." What we'll discuss here are the back-end API endpoints that are implemented in Visual Studio and deployed to Azure App Service. The App Service location in the architecture is indicated in Figure 4-1.

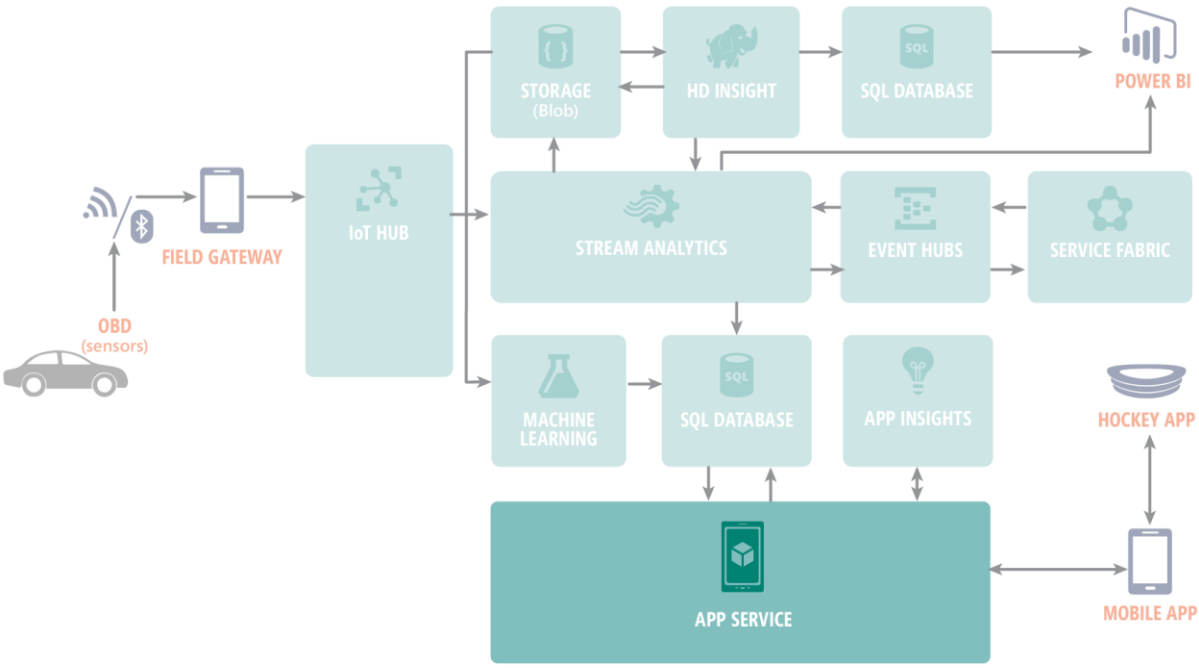


Figure 4-1: App Service placement in the MyDriving architecture

Apart from the data being sent from the IoT field gateway to the cloud gateway—roles played in MyDriving by the app and the IoT hub in the backend, respectively—the primary external interface to the back end is the set of API endpoints that are hosted in App Service.

Note Some Azure storage accounts in the MyDriving system, such as SQL databases, are also accessible externally. Visualizations in Power BI make use of these connections, as described in Chapter 6 in the "Power BI" section.

It's worth noting at the outset that you can create a REST API with a number of technologies in Azure. You can build a completely custom server on a virtual machine, for example, or you can use Azure Cloud Services for full control over the software without dealing with VM infrastructure. Many mobile apps, however, have very similar back-end requirements for storage, push notifications, authentication, and API endpoints. The Mobile Apps feature of Azure App Service thus provides a number of pre-built, mobile-specific capabilities.

We've already seen, in fact, how the MyDriving app uses a number of these. In this section, then, we'll begin with a primer for helping you understand Mobile Apps in general. We'll then dive into the general process of creating endpoints in Visual Studio with ASP.NET Web API and deploying that code to App Service, which is how we created our endpoints for MyDriving. We'll finish by looking at the MyDriving API code itself, which is found in the src/MobileAppService folder on GitHub. (A reference for the APIs can be found in Chapter 10, *Reference*.)

Primer: App Service for mobile back ends

The best mobile apps—those that best engage consumers and thus perform well for their publishers—are consistently those that have great back ends to power the experience. Many apps, for example, clearly rely on data retrieved from a back end that could be collecting and processing information from a variety of other sources. In fact, it's often helpful to think of such apps (especially in branded and enterprise scenarios) as turning a mobile device into a viewport onto the cloud. Even games often use back-end services to roam scores and game state across devices, manage leaderboards, and alert players to new challenges.

A back end in the cloud, accessed through HTTP requests or other such protocols, is also by nature shared across all potential clients and independent of client platforms like iOS, Android, and Windows. When combined with cross-platform app technologies like Xamarin and Apache Cordova, a large portion of your total application—back end plus clients—is client platform-neutral. What's more, you can update your back-end code any time without going through specific app store submissions.

Across the whole gamut of cloud-connected mobile apps, there is a core set of back-end capabilities that many apps require:

- **Push notifications** let the back end send messages to any number of client devices, independent of specific mobile platforms. Push notifications can be broadcast generally or targeted to an individual user's devices or even a single device. (This is an entirely different matter from sending commands to IoT devices, mind you!)
- **Cloud storage** easily shares data between mobile apps and the back end, and handles local offline caching and synchronization to accommodate the varied connectivity on mobile devices. Data can be organized on per-user, per-app, and global bases. The back end can attach business logic to operations like insert, update, and delete. This makes it possible, for example, to detect changes made by one user or client that then generate push notifications to a group of others.
- **Authentication** provides enterprise-grade user authentication through mechanisms like Azure Active Directory and OAuth providers such as Microsoft, Google, Facebook, and Twitter. A unique identity for every user enables the back end to send personal push notifications and partition user data so that it's available in the same app on multiple devices.

Building on this, many mobile apps also need a few other capabilities:

- **Background jobs** do continuous and/or scheduled processing in the always-on and always-connected back end. They're commonly used to retrieve, process, and cache data from other services so they're immediately available to the mobile app. By centralizing such processing in the back end, you minimize power and data usage on the mobile devices and avoid having to deal with sporadic mobile connectivity. Collecting data in the back end can also minimize the number of requests to other services that might impose their own throttling limits.
- **Custom REST APIs** define endpoints that can be called by an app to invoke operations that aren't strictly tied to data access. These include registering a user or device with a service, triggering business logic, sending notifications to another user, or running complex queries that aren't easily done on the client.

Additionally, you often want both mobile and web experiences to be connected to the same back end—especially for shared data. For enterprise apps especially, you might need to connect to on-premises resources and tie into other business processes.

The question, then, is how to build up a back end with capabilities that will also easily scale up and down with customer demand. It's certainly possible, of course, to build all these from scratch inside virtual machines, but like many developers you're probably much more interested in delivering great user experiences to your mobile customers than you are in building and maintaining custom infrastructure.

Mobile Apps in Azure App Service

Recognizing the needs described in the previous section, Microsoft has packaged features together in the platform-as-a-service (PaaS) offering called Mobile Apps, which you can use for free until you're ready to ramp up toward production and then scale out from there.

Mobile Apps is one of three distinct "app types" that are all part of Azure App Service: Web Apps, Mobile Apps, and API Apps. All of these are built on a common web-based runtime and share the same UI in the Azure portal. This blurs the distinction between the types, but in general, it's best to simply focus on the features you need instead of the types. For mobile scenarios, it's best to create a "Mobile App" and then use one of the options under **Tools > Quick Start** to get starter templates for the client code (options include native apps along with Xamarin and Cordova).

Mobile Apps provides a variety of features that you can turn on individually by configuring them in the Azure portal, including those in the following table.

Feature	Location in the Azure settings UI	Description
Authentication	Features > Authentication / Authorization	Provides authentication using Microsoft, Google, Facebook, Twitter, or Azure Active Directory identity providers, and authorization to control access to back-end features.
Storage	Mobile > Easy tables	Works with storage through in-memory objects that use Azure SQL databases or table storage. The Azure libraries you include with the app take care of all the HTTP requests to the back end to keep the local and cloud copies in sync. For scenarios with variable connectivity, you can use storage with offline data sync as well.

Push notifications	Mobile > Push	Uses Azure Notification Hubs to send notifications easily to iOS, Android, and Windows devices, which can scale from a single device to millions of devices.
REST APIs	Mobile > Easy APIs	Creates the necessary Node.js infrastructure to expose and manage API endpoints, leaving you to focus on the core API implementation that you can do directly in the Azure portal. Alternately, you can use ASP.NET Web APIs with C# to create API endpoints. In this case, you start with a template in Visual Studio and then publish the code to the App Service as described later in "Creating an App Service API project in Visual Studio."
Web Jobs	Web Jobs > Webjobs	Uploads code files that run as background tasks in the App Service. You can also run them manually through the portal.

Note Azure App Service is the next-generation Azure product that combines the capabilities of older products such as Azure Websites and Azure Mobile Services, among others. You'll occasionally see references to these older products in code. The client-side library for working with Mobile Apps, for example, is still called the Azure Mobile Services SDK, and the main class you use in that library is `MobileServicesClient`.

Additional notes

Generally speaking, when you commit a Mobile App in Azure App Service to a technology like Node.js using Easy APIs or Easy Tables, then any Web App you might create in the same App Service must also be built with Node.js. Similarly, if you start with a Web App built on ASP.NET, you can also deploy endpoints built with ASP.NET Web APIs to the same App Service, but you can't use Easy APIs and Easy Tables because Node.js won't be there. (The Azure portal will say "Unsupported service" if you try to configure these features in an App Service that's already set up for ASP.NET.)

In short, avoid trying to mix and match technologies within the same App Service. Instead, simply create another Web App or Mobile App in a different App Service, which you can then configure independently.

Don't worry about incurring higher costs by doing this. The scaling unit for your pricing tier within Azure is called a *plan,* and you can create multiple App Services within the same plan. For details, see the Azure App Service plans in-depth overview.

Furthermore, it's not a problem to share data between these different App Service instances. Azure SQL databases and Azure table storage are not bound to an App Service instance, which means that any number of App Service instances running in any number of plans can all share the same database. This database can also be accessed by other Azure services in your overall solution.

In the MyDriving project, for example, a single Azure SQL database is shared across the App Service instance that implements the APIs, the Stream Analytics that processes data from the IoT hub, and the Machine Learning component. As you extend this sample in your own projects, you can make use of this same central store for other needs.

Creating an App Service API project in Visual Studio

As described in the previous section, the two API options in Azure App Service work a little differently. "Easy APIs" with Node.js is handled directly through the Azure portal, or by doing work locally and uploading via FTP. Implementing APIs with C#/ASP.NET, on the other hand, involves a workflow that instead begins in Visual Studio and uses ASP.NET Web API.

In this section, we'll follow that workflow, because although deploying your own system with the MyDriving Resource Manager template creates the back-end App Service, the template doesn't create the API endpoint code that you need to deploy to that service from Visual Studio.

> **Note** For a more complete walkthrough, including steps you need to take on the Azure portal, see Work with the .NET back-end server SDK for Azure Mobile Apps.

The workflow begins in Visual Studio with **File > New Project.** Select the **C# > Cloud > Azure Mobile Service** template as shown in Figure 4-2.

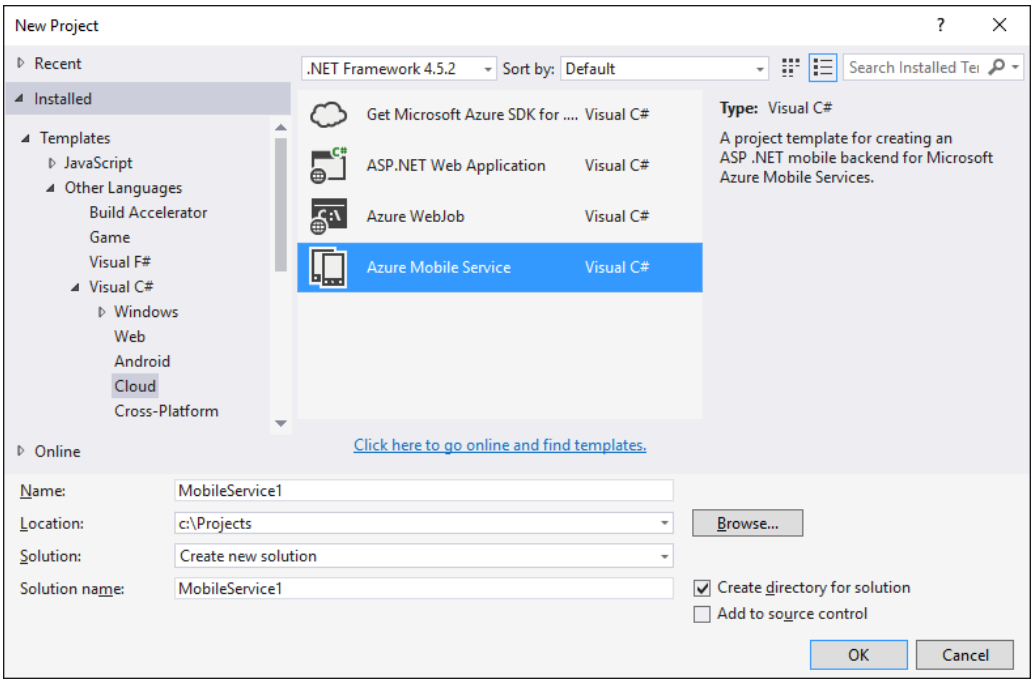


Figure 4-2: Creating a new App Service API project in Visual Studio

Choosing this template brings up a second dialog with a more options. Notice that the **Web API** option for folders and core references is selected automatically, which is what you want (Figure 4-3 on the next page).

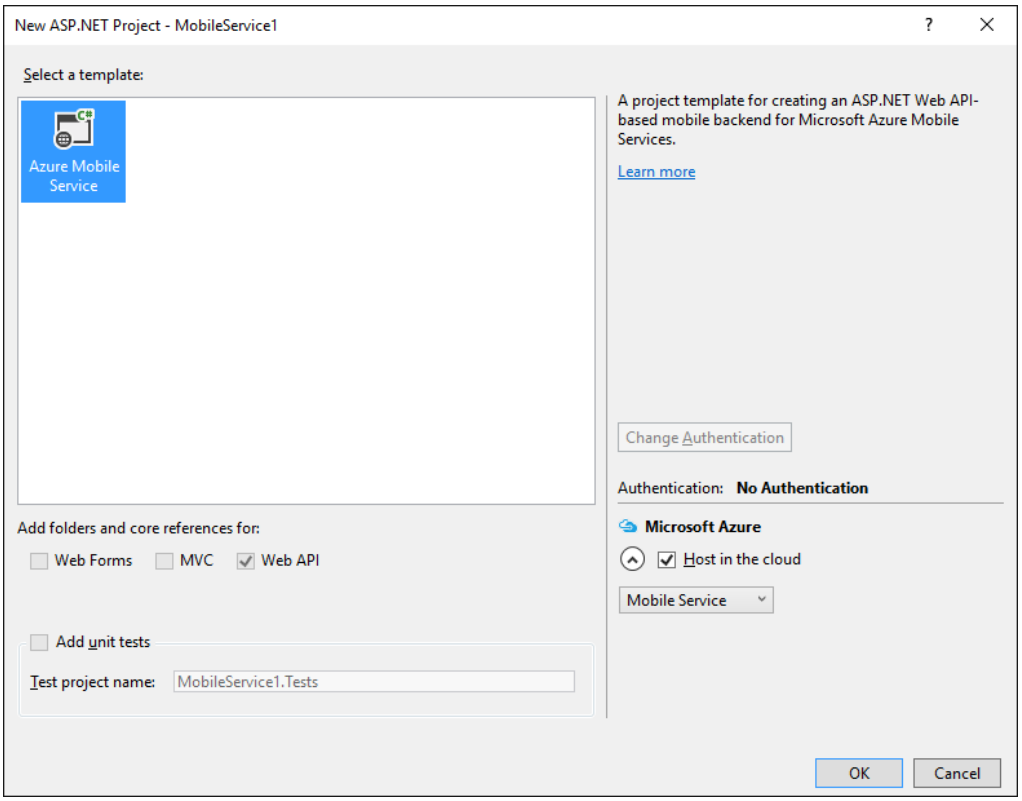


Figure 4-3: Options for an App Service API project in Visual Studio

Clicking OK brings up a **Create Mobile Service** dialog in which you can create and provision an instance of Azure Mobile Services if you want. However, Mobile Services is the older version of App Service, and currently it's better to create the App Service instance separately (as the Resource Manager template does for MyDriving). So just press **Cancel** here. You can create the App Service later on too.

In any case, this Visual Studio template produces a project structure that's oriented around the ASP.NET Web API as well as Entity Framework Code First for data access. This is similar to the standard ASP.NET MVC architecture that's often used to build web apps, but because we're implementing only API endpoints, we don't have any views to speak of, just the model and controllers.

Here are the core files in the project that concern us here:

File	Description
DataObjects/ToDoItem.cs	The data object classes; the template creates a single `TodoItem` class.
Models/MobileServiceContext.cs	The data model context derived from the Entity Framework `DbContext` class and containing properties for each public member of the data model. In the context class (`MobileServiceContext` by default), the value "MS_TableConnectionString" is passed to the `DbContext` constructor to allow Entity Framework to establish the database connection. More on this shortly.
Controllers/TodoItemController.cs	A default API implementation (*tables/TodoItem*) with GET, PATCH, POST, and DELETE operations.

Global.asax, Global.asax.cs	The default (and replaceable) startup object and application entry point.
App_Start/WebApiConfig.cs	The code that's called from the application entry point to do startup configuration and seed the database if needed.
Web.config	The typical ASP.NET configuration file.

In conjunction with Azure App Service, there are two very important details to understand:

- When you set up an App Service instance on the Azure portal, you typically configure a data connection through the **Data Connections** setting. If you use a name that's also in Web.config, such as `MS_TableConnectionString` (as shown in Figure 4-4), it *overrides* any connection string in Web.config with the same name when the code is running inside App Service. In this way, you can use the default ASP.NET database for local debugging without having to change any configuration when you deploy to App Service.

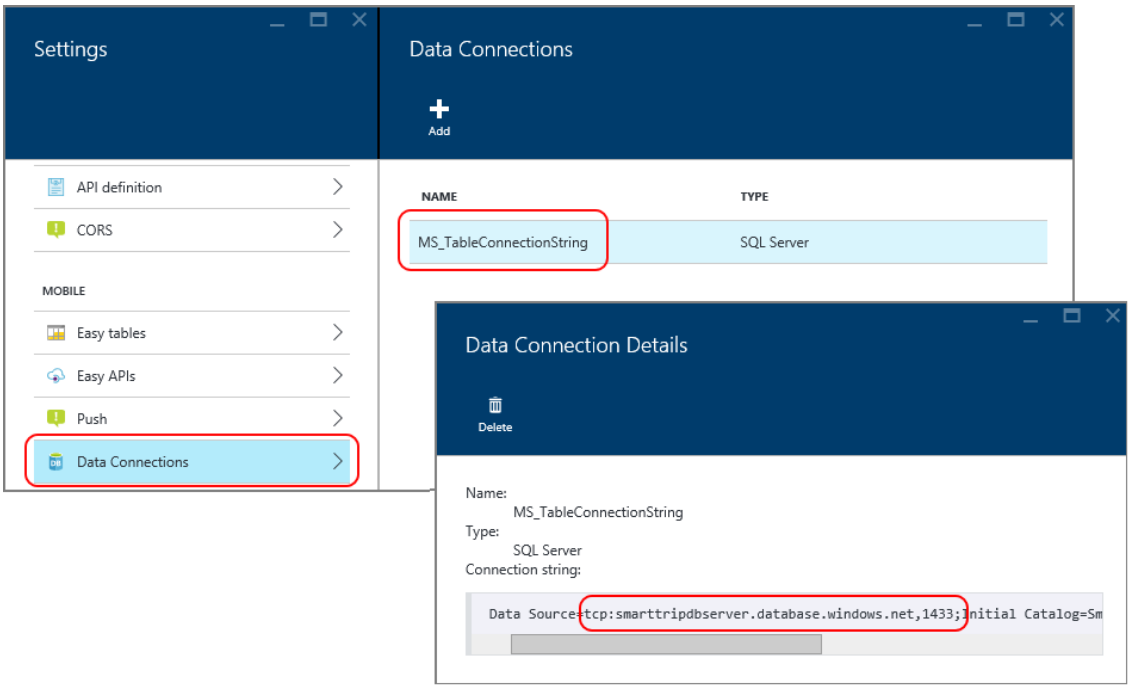


Figure 4-4: Finding the connection string for App Service

- In the template-generated project (and in MyDriving) you won't find any explicit code or configuration that names the API endpoints and routes them to code. ASP.NET Web API instead generates routing from the names that are used for your controller classes and/or special attributes on those classes. You can find details on Routing in ASP.NET Web API, but the short of it is that a class called `<name>Controller` (which is derived from `TableController<T>`) produces routing for the endpoint *tables/<name>*. You can see how this works in the following declaration in TodoItemController.cs:

```
public class TodoItemController : TableController<TodoItem>
```

- This declaration results in an endpoint *tables/TodoItem*. Note that "TodoItem" is case-insensitive; *api/todoitem* works just fine. Similarly, if a class called `<name>Controller` derives from `ApiController`, you'll get a routing for *api/<name>*. We'll see examples in

the next section.

It's easy to test these locally: press **F5** in Visual Studio to launch the application in a browser, and append the routes like */tables/ToDoItem* to the localhost URL to invoke the API. You can also set breakpoints in Visual Studio and step through the implementations.

The last question before we look at the MyDriving specifics is how we get our API code in Visual Studio deployed to the App Service. It's simple. First select the **Publish...** command from the project's context menu as shown in Figure 4-5, which brings up the **Publish Web** dialog box in Figure 4-6.

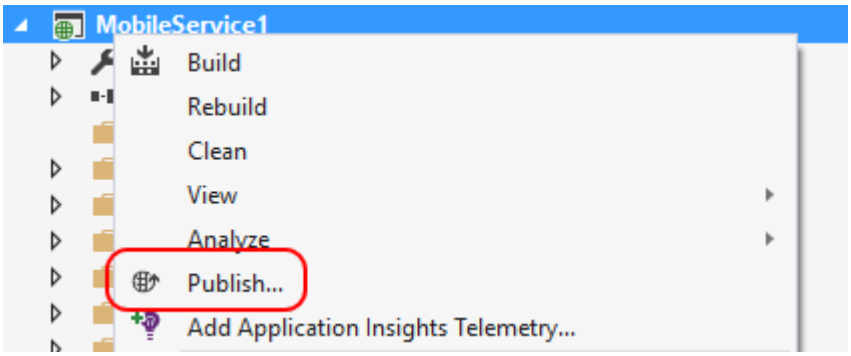


Figure 4-5: The Publish command on the project's context menu

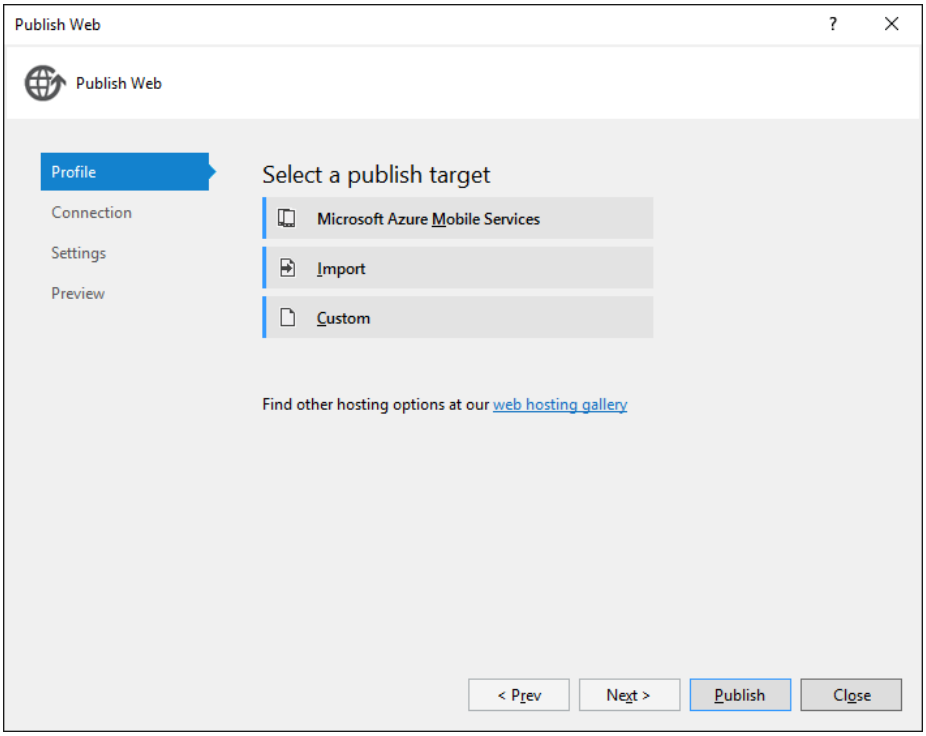


Figure 4-6: The Publish Web dialog box

Select **Microsoft Azure Mobile Services**, and then click **Next**. You'll be prompted to select an existing App Service from your Azure account, or to create a new one as shown in Figure 4-7 (next page).

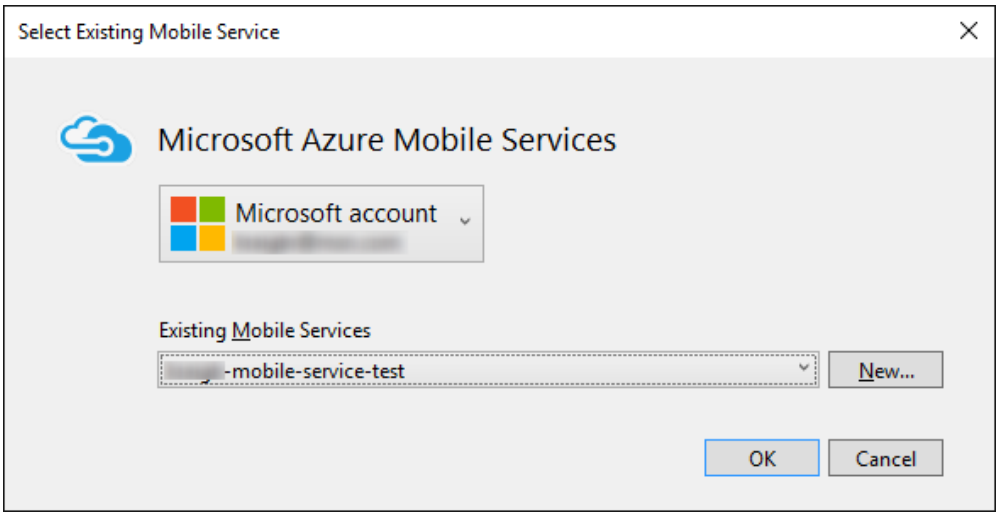


Figure 4-7: Selecting an Azure account and an App Service (or Mobile Service) in that account

When you click **OK**, Visual Studio downloads all the necessary publishing settings from Azure that appear in the **Connection** tab of the **Publish Web** dialog box, shown in Figure 4-8.

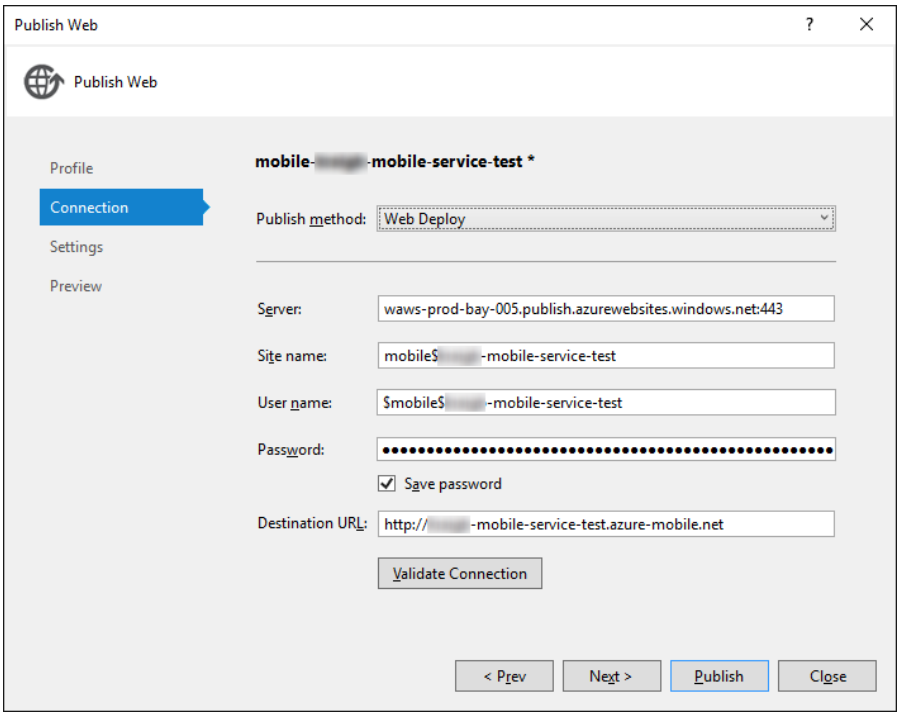


Figure 4-8: The connection information for App Service brought into the Publish Web dialog box in Visual Studio

Review these settings along with everything on the other tabs in the dialog box. When you're ready to deploy, just select **Publish**. Visual Studio then builds the project and deploys the application and all dependent NuGet packages to Azure. From that point on, you can call APIs with the App Service's URL (the destination URL that's shown in the **Publish Web** dialog box).

App Service in MyDriving: storage and data model

In MyDriving, App Service provides for both API endpoints and cloud storage that's easily accessible from both the back end and the client app. Because the APIs generally draw from storage, we'll look at that aspect of App Service first.

Storage is handled through Azure tables (backed by a SQL database and including offline sync), as we've already described in Chapter 3, *The Mobile App*. The app simply uses methods and properties in the `MobileServiceClient` object to store and retrieve data. That object, meanwhile, transparently handles HTTP requests to make sure that changes made in the client are replicated to the back end, and that changes in the back end are replicated to the client. This makes working with cloud storage as simple as it is to work with a local database. For example, once we have an object for the table in hand, we can use LINQ syntax to do a direct query:

```
var items = await Table.Where(s => s.Id == id).ToListAsync().ConfigureAwait(false)
```

Now let's switch over to the API implementations themselves in src/MobileAppService in the repository. In this Visual Studio solution, the MyDriving.DataObjects shared project is loaded directly from the MobileApps folder, so that we're always working with the same data model as the Xamarin app. The MobileAppService project, which is where we'll be working for the rest of this chapter unless noted, has the structure that was provided by the Visual Studio template described in the previous section. In general, this type of project is also referred to as an ASP.NET Web API project.

In the MobileAppService code, the data context for the APIs is the `MyDrivingContext` class, derived from the Entity Framework `DbContext`. Here, in fact, is the bulk of Models/MyDrivingContext.cs (slightly edited):

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration.Conventions;
using System.Linq;
using Microsoft.Azure.Mobile.Server.Tables;
using MyDriving.DataObjects;

namespace MyDrivingService.Models
{
    public class MyDrivingContext : DbContext
    {
        // MS_TableConnectionString is defined in the App Service
        // to use an Azure SQL database.
        private const string ConnectionStringName =
            "Name=MS_TableConnectionString";

        // Let the EF DbContext constructor make the connection
        public MyDrivingContext() : base(ConnectionStringName)
        {
        }

        // Define the data model using classes in MyDriving.DataObjects
        public DbSet<Trip> Trips { get; set; }
        public DbSet<TripPoint> TripPoints { get; set; }
        public DbSet<UserProfile> UserProfiles { get; set; }
        public DbSet<IOTHubData> IOTHubDatas { get; set; }
        public DbSet<POI> POIs { get; set; }
    }
}
```

From this point on, all the controllers that implement the APIs can just create an instance of `MyDrivingContext` to talk to storage as needed:

```
MyDrivingContext context = new MyDrivingContext();
```

Let's now look at those implementations.

App Service in MyDriving: API controllers

The MyDriving back end exposes the following APIs (the complete details are in Chapter 10). You'll find the respective code files in the project's Controllers folder:

- Tables (GET, PATCH, POST, DELETE; GET supports both all items and a single item)
    - *tables/IOTHubData* (IOTHubDataController.cs)
    - *tables/POI* (POIController.cs, GET only)
    - *tables/Trip* (TripController.cs)
    - *tables/TripPoint* (TripPointController.cs)
    - *tables/UserProfile* (UserProfileController.cs)
- IoT device provisioning (GET/POST; see Chapter 3 for use by the app)
    - *api/Provision* (ProvisionController.cs),
- User information (GET; see Chapter 3 for use by the app)
    - *api/UserInfo* (UserInfoController.cs)

The tables, each represented by a `DbSet` property in the context object, have `<name>Controller` classes that derive from `TableController<T>` as described previously. `IOTHubDataController` and `TripPointController` are both stock implementations that support all the basic operations through bits of code that calls Entity Framework. The GET operations, for example, can be done in one line:

```
// api/TripPoint from TripPointController.cs
[Authorize]
public IQueryable<TripPoint> GetAllTripPoints()
{
    return Query();
}

[Authorize]
public SingleResult<TripPoint> GetTripPoint(string id)
{
    return Lookup(id);
}

// api/POI from POIController.cs
[Authorize]
public IQueryable<POI> GetAllPOIs(string tripId)
{
    return Query().Where(p => p.TripId == tripId);
}
```

Note The `Authorize` attributes you see here indicate that the scope of the query is always limited to the data for the current authenticated user, and that queries are not allowed without authentication. For more details, see Authentication and Authorization in ASP.NET Web API.

The `Trip` and `UserProfile` tables are a little more complex because their data objects contain one or more collections:

- `Trip` (MyDriving.DataObjects\Trip.cs) contains a collection of `TripPoint` objects, which store snapshots of OBD and GPS data.
- `UserProfile` (MyDriving.DataObjects\UserProfile.cs) contains a string collection called `Devices` that contains a user's registered IoT devices.

In such cases you usually want to support queries on those collections. To do this, MyDriving uses a custom attribute called `QueryableExpand` that's implemented in Helpers/QueryExpandAttribute.cs:

```
public class QueryableExpandAttribute : ActionFilterAttribute
{
    // Class overrides the OnActionExecuting to implement the attribute's
    // behavior based on the URL query string.
}
```

By deriving from `ActionFilterAttribute`, the name of the class minus the "Attribute" suffix (which is, in fact, optional), becomes the name of the attribute. A class name of `ExpandableQueryAttribute` (or just `ExpandableQuery`) would create an attribute called `ExpandableQuery`.

With this attribute in place, we can now adorn the GET operation in TripController.cs:

```
[QueryableExpand("Points")]
[Authorize]
public SingleResult<Trip> GetTrip(string id)
{
    return Lookup(id);
}
```

and in UserProfileControllers.cs:

```
[QueryableExpand("Devices")]
[Authorize]
public async Task<IQueryable<UserProfile>> GetAllUsers()
{
    var id = await IdentitiyHelper.FindSidAsync(User, Request);
    return Query().Where(s => s.UserId == id);
}
```

Such a custom attribute is a very concise way to attach reusable extra behaviors to any number of APIs without cluttering each controller with duplicate code.

For the two non-table APIs, their controllers derive from `System.Web.Http.ApiController`. By convention, a class called `<name>Controller` creates a routing for *api/<name>*:

```
// ProvisionController.cs
[MobileAppController]
public class ProvisionController : ApiController
{
    // ...
}

// UserInfoController.cs
[MobileAppController]
public class UserInfoController : ApiController
{
    // ...
}
```

You'll notice here that these classes are marked with the `Microsoft.Azure.Mobile.Server.MobileAppController` attribute. This ties into the project's startup code, which follows the steps in the "Initialize the server project" section of Work with the .NET back-end server SDK for Azure Mobile Apps. The startup code creates an instance of the `Microsoft.Azure.Mobile.Server.MobileAppConfiguration` object and calls either its `MapApiControllers` or `UseDefaultConfiguration` method (the latter calls the former). The MobileAppService project does this in App_Start/Startup.MobileApp.cs:

```
public static void ConfigureMobileApp(IAppBuilder app)
{
    // [...]

    new MobileAppConfiguration()
        .UseDefaultConfiguration()
        .ApplyTo(config);

     // [...]
}
```

What's left are the API implementations that respond to the various HTTP requests we want to support—namely, controller methods that are adorned with attributes like `[HttpGet]` and `[HttpPost]`, or that follow the ASP.NET Web API naming conventions. Both options are described on Routing in ASP.NET Web API. We use attributes in MobileAppService.

For example, the method that implements GET for *api/UserInfo* is as follows, relying on naming conventions (UserInfoController.cs):

```
[MobileAppController]
public class UserInfoController : ApiController
{
    // GET api/UserInfo
    [Authorize]
    public async Task<DataObjects.UserProfile> Get()
    {
        // [Omitted: retrieve details for the currently authenticated user from
        // the identity provider. This sets variables named first, past, and
        // profile (a URL) from the identity provider's info.]

        // Add the user if not already in storage
        var context = new MyDrivingContext();
        var curUser = context.UserProfiles.FirstOrDefault(u => u.UserId ==
             userId);

        if (curUser == null)
        {
            curUser = new MyDriving.DataObjects.UserProfile
            {
                UserId = userId,
                ProfilePictureUri = profile,
                FirstName = first,
                LastName = last
            });

            context.UserProfiles.Add(curUser);
        }
        else
        {
            curUser.FirstName = first;
            curUser.LastName = last;
            curUser.ProfilePictureUri = profile;
        }

    await context.SaveChangesAsync();
    return curUser;
    }

    // [Other class members omitted]
}
```

The *api/provision* endpoint—the one we used from the mobile app (see Chapter 3) to register the IoT device with the back end—is implemented in ProvisionController.cs and uses the `HttpGet` attribute. (Details are replaced by comments in this code.):

```
[MobileAppController]
public class ProvisionController : ApiController
{
    // GET api/provision
    [HttpGet]
    [Authorize]
    public async Task<IEnumerable<Device>> Get()
    {
        // [Return a list of the user's registered devices]
    }

    // POST api/provision
    [HttpPost]
    [Authorize]
    public async Task<IHttpActionResult> Post(string userId, string deviceName)
    {
        // [Make sure the user is authenticated]

        // [Check whether we're at the maximum number of devices]

        // [Register the device and return the registration]
    }
}
```

And that sums up the back end API implementations! Again, this code must be manually deployed to the back end's App Service. If you've deployed all the Azure services by using the MyDriving Resource Manager template, you'll need to open the MobileAppService solution in Visual Studio and manually publish it to the App Service that was created in that process. See the earlier section, "Creating an App Service API project in Visual Studio," for publishing instructions.

Using and testing the API

Because all the APIs in MyDriving require authorization, any calls to them (once deployed to Azure App Service) must be done by an authenticated user. That is, callers need to authenticate with App Service first before the other APIs are available. For an unauthenticated service, however, you can always test endpoints directly from a browser.

To test APIs from a mobile app, it's easiest to use the Azure Mobile Services client SDK like we saw with the MyDriving Xamarin app in Chapter 3. Remember that the client SDK provides an object model that's automatically mapped to the *tables/** API, eliminating the need to make HTTP requests directly.

For other endpoints, such as *api/provision*, the SDK's `MobileServiceClient` object provides a generic method to make the necessary HTTP requests. To return to the Xamarin app project for a moment, take a look again at `ProvisionDevice` in MyDriving.AzureClient/DeviceProvisionHandler.cs:

```
public async Task<string> ProvisionDevice()
{
    // [Some error checking omitted]

    Dictionary<string, string> myParms = new Dictionary<string, string>();
    myParms.Add("userId", Settings.Current.UserUID);
    myParms.Add("deviceName", Settings.Current.DeviceId);

    var client = ServiceLocator.Instance.Resolve<IAzureClient>()?.Client
        as MobileServiceClient;

    try
    {
        var response = await client.InvokeApiAsync("provision", null,
            HttpMethod.Post, myParms);
        AccessKey = response.Value<string>();
    }
    catch { /* ... */ }
```

```
        // The DeviceConnectionString is created with the AccessKey (see below)
        Settings.Current.DeviceConnectionString = DeviceConnectionString;
        return Settings.Current.DeviceConnectionString;
    }

    public string DeviceConnectionString
    {
        get
        {
            string connectionStr = String.Empty;
            if (!String.IsNullOrEmpty(AccessKey) && !String.IsNullOrEmpty(HostName)
                && !String.IsNullOrEmpty(DeviceId))
            {
                // HostName is set to "mydriving.azure-devices.net" by default
                connectionStr =
                    $"HostName={HostName};DeviceId={DeviceId};SharedAccessKey={AccessKey}";
            }

            return connectionStr;
        }
    }
```

As you can see, `MobileServiceClient.InvokeApiAsync` takes the name of the API, the HTTP method we want to invoke, and the necessary parameters for that request. This invocation clearly ends up in the `ProvisionController.Post` method we just saw in the previous section, which does the necessary work and returns the registration key. Enough said!

Of course, you'll probably want to do more extensive work like unit testing and load testing in your own deployments and customizations of the MyDriving MobileAppService project. For more about those subjects, see the following resources:

- Testing and debugging ASP.NET Web API
- Load test your app in the cloud with Visual Studio and Visual Studio Team Services
- Unit testing of Web API from Visual Studio (C# corner)
- Load testing of Web API from Visual Studio (C# corner)

CHAPTER 5

DevOps

In the MyDriving project, we operate a rapid DevOps cycle to build and distribute the app, get feedback about how it performs and what users do with it, and then use that knowledge to feed in to further development cycles. To monitor usage and performance, we get telemetry from both the client and server components of the application, as well as feedback from the users themselves.

Some releases have restricted distribution to designated testers; we have also organized *flighting* (tests of new features with restricted audiences), and *A|B testing* (parallel tests of alternative UI).

Managing distributions and integrating monitoring over multiple client and server components isn't a trivial task. This process is an essential part of the architecture of the application: we can't create a system of this kind without an iterative development cycle and good monitoring tools.

In this section, then, we'll share how we manage our DevOps cycle with a variety of tools. First, a couple of simple matters:

- Our code is kept in a GitHub repository: https://github.com/Azure-Samples/MyDriving, where we accepts contributions, of course! Different members of our team worked with the repository through Visual Studio, others through the command line or GUI tools.
- The work we did on the Xamarin app (Chapter 3, *The Mobile App*) happened in both Visual Studio, which supports Windows and Android, and Xamarin Studio which supports iOS and Android. Work on the API endpoints for the back end happened in Visual Studio, as already described in Chapter 4, *App Service API Endpoints*.

That leaves the larger issues of running builds, running tests, managing releases and deployment, and monitoring the mobile app and the back-end App Service. Let's look at each in turn and see how they employ the following:

- Visual Studio Team Services for cloud-based build, testing, and deployment services for both client and server.
- HockeyApp for test management, crash reporting, and usage analytics of the mobile app.
- Xamarin Test Cloud for testing of the mobile app on a variety of devices.
- Application Insights for detailed analytics of the App Service.

Mobile app DevOps

For the Xamarin mobile app, we've set up builds and testing on Visual Studio Team Services for continuous integration. This means that every code commit triggers a build for all platforms—Android, iOS, and Windows—and runs tests on each. The results of all these tests are shown on the project's dashboard (Figure 5-1).

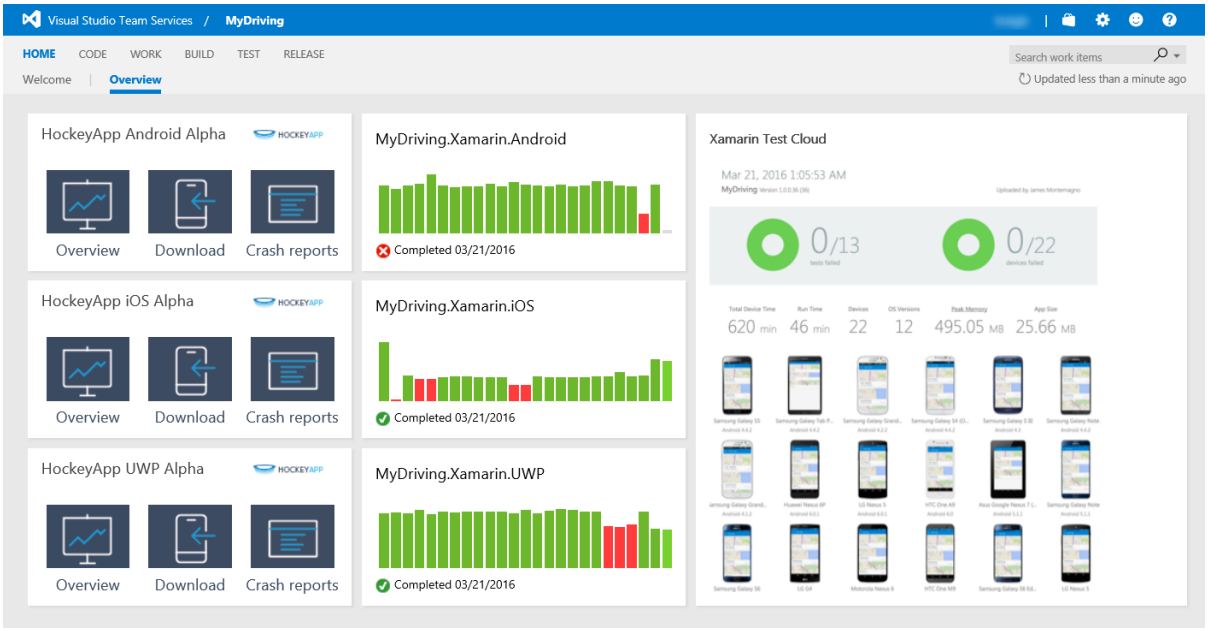


Figure 5-1: The Visual Studio Team Services dashboard for the MyDriving mobile app

When the app packages successfully pass all tests, they are then published to a HockeyApp private app gallery, from where testers can acquire them. Testers also receive notifications that updates are available.

The app itself is instrumented to gather and report both user feedback and telemetry using the HockeySDK. HockeyApp also automatically gathers crash reports that include device type, OS version, the exception message and location, and user and session counts. It feeds these, along with user feedback and screenshots, into the project's backlog as work items for the next cycle as shown in Figure 5-2 on the next page.

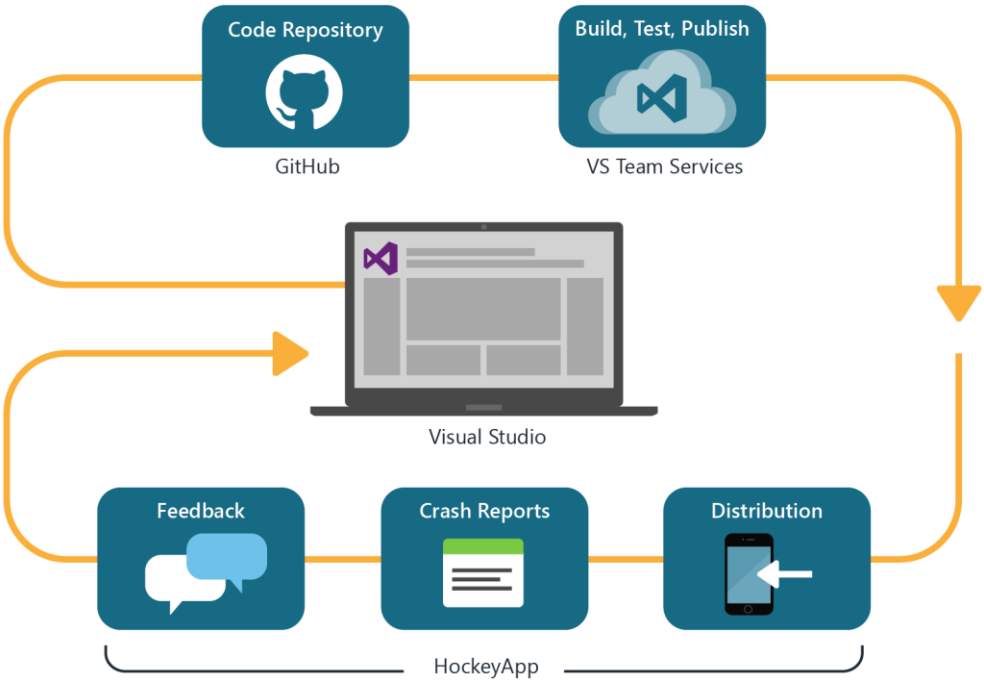


Figure 5-2: Mobile DevOps with HockeyApp

Visual Studio Team Services build definitions

Let's take a closer look at the build definition MyDriving.Xamarin.Android in Visual Studio Team Services. Build definitions for the other platforms are very much the same; you can find complete descriptions in Chapter 10, *Reference*, and we'll provide a few notes for iOS shortly.

> **Note** In src/MobileApps on GitHub you'll see three solution files: MyDriving.sln, MyDriving.XS.sln, and MyDriving.iOS.sln. MyDriving.sln is what you normally open in Visual Studio to work with the app code. MyDriving.XS.sln is for Xamarin Studio and is also used to build on Visual Studio Team Service. MyDriving.iOS.sln is used exclusively for builds that use Visual Studio Team Services.

First of all, we've set up the definition for continuous integration as shown in Figure 5-3, so that builds run with every commit to the source repository.

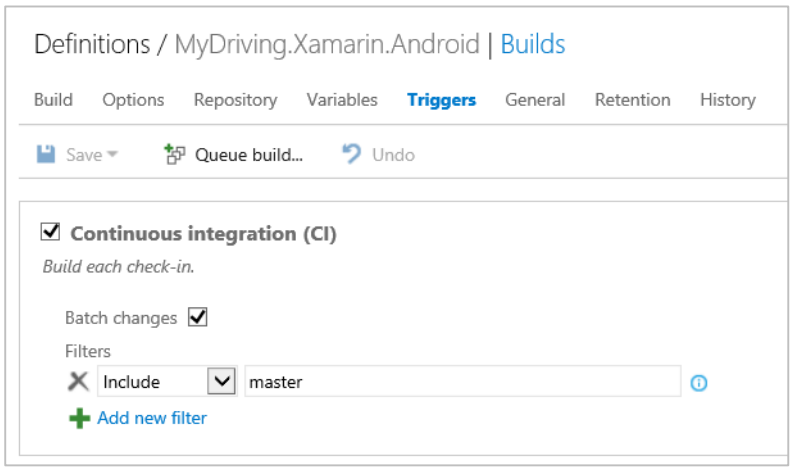
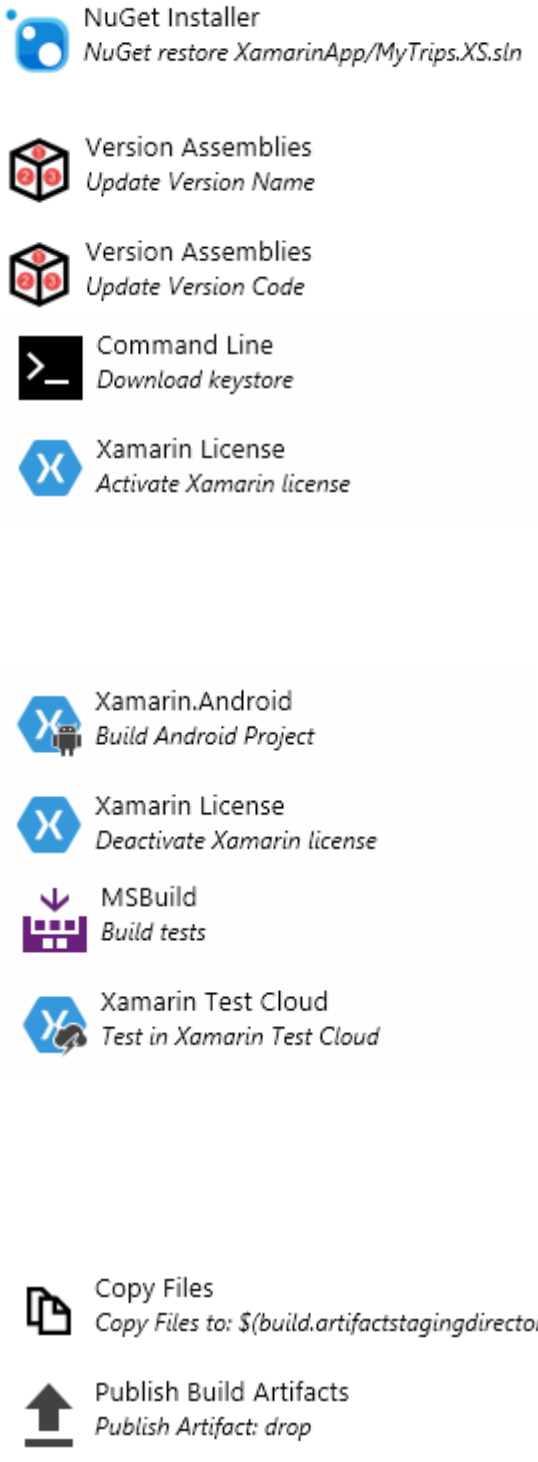


Figure 5-3: Configuring a build definition for continuous integration

The build definition is composed of the following steps, which are run in the order shown:



The first step restores NuGet components to the solution, so that we don't have to keep them in the GitHub repository.

Version Assemblies is a useful utility that updates the version number of the built assembly to match the build. It's one of many third-party plug-ins available in the Visual Studio Marketplace.

A Xamarin license supports use on five simultaneous machines. When running builds in the cloud, it's necessary to activate the license on the machine used by the build agent. Once the build is complete, it's then critical to deactivate the license on that build agent, otherwise you'll quickly fill your quota. Rather than keep the keys in source control with the code, we keep them in a key store. The first step copies them to the local source folder for the activation step.

The Xamarin build plug-in does the real work. Immediately after building, we deactivate the license from the current build machine.

Next we build and run the tests. This can include unit testing as well as automated UI testing. In MyDriving we're demonstrating UI tests only, which are run on multiple physical devices in the Xamarin Test Cloud (see Figure 5-4 and Figure 5-5 on the next page). The tests, written in C# with the NUnit test framework, are located in the MyDriving.UITests project of the Xamarin app solution. For details on writing these types of tests, see the Introduction to Xamarin.UITest documentation.

After the build is complete, app packages are published to a staging area so they can be downloaded or used by Release Management definitions.

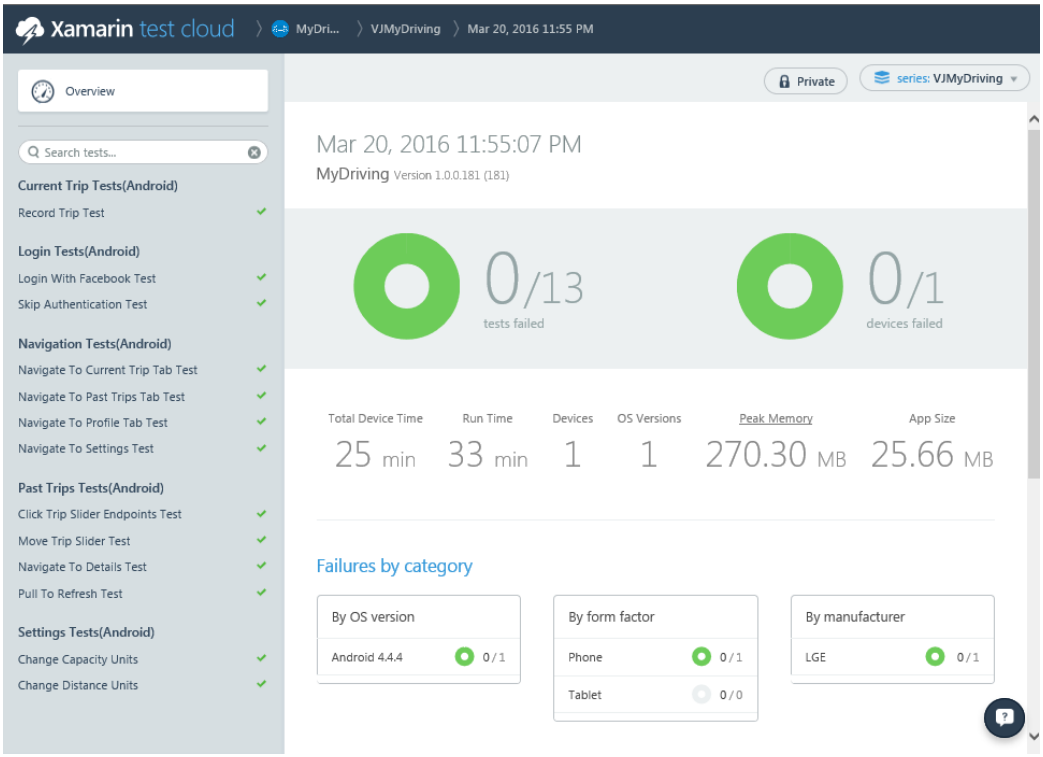


Figure 5-4: The Xamarin Test Cloud dashboard.

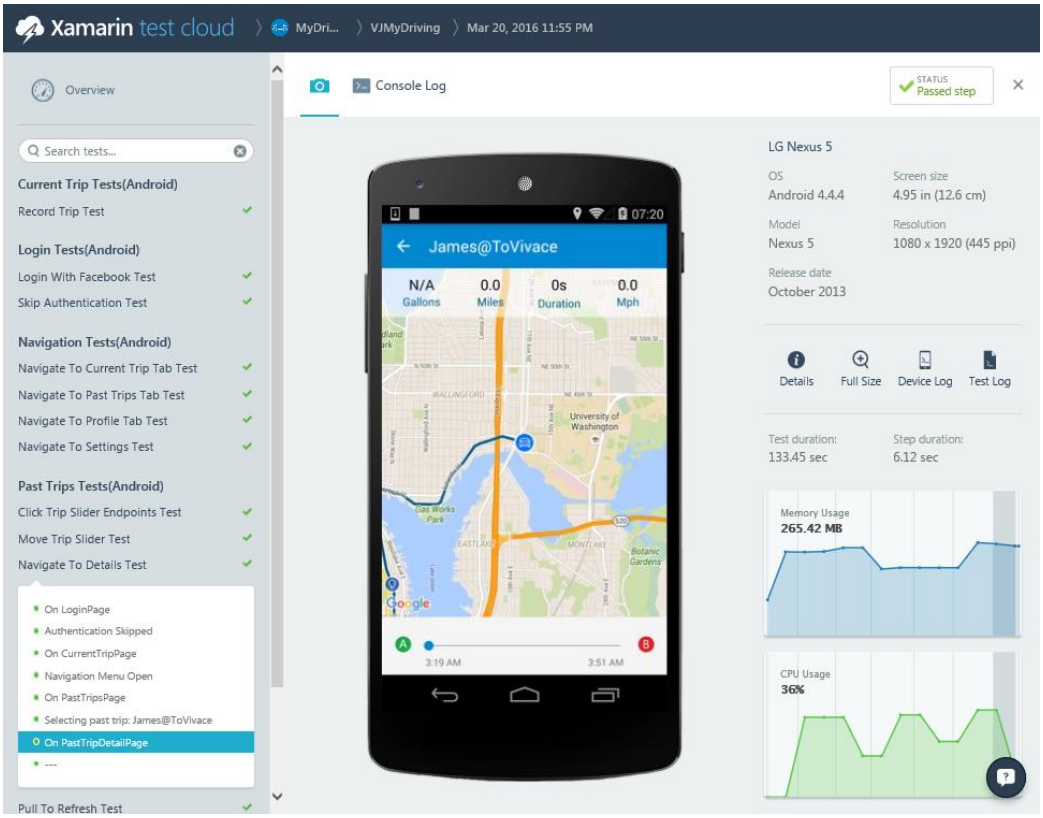


Figure 5-5: Xamarin Test Cloud showing results for the On PastTripDetailPage step of the Navigate To Details Test

Building for iOS with MacinCloud

Building an iOS app always requires a Mac OS X machine. This requirement isn't satisfied by a Visual Studio Team Services build agent running in a Windows virtual machine. The iOS build definition, then, employs MacInCloud, a service that provides for-rent Mac machines for building and testing. (You can also use your own Mac for these purposes.) For more details, see MacinCloud Visual Studio Team Services Build. Also see Visual Studio Team Services Build Agent Plan (macincloud.com) for pricing and machine configuration.

After you've set up a machine in MacinCloud, your iOS builds with the *xcode* demand will be assigned to that machine. You can also have Mac build agents on their own agent queue, and use that queue for the builds. To do this, edit the build definition, open the **General** tab, and select the appropriate queue in the **Default agent queue** drop-down as shown in Figure 5-6. To manage queues and make sure the machine is assigned to the right queue, click the **Manage** link to opens the Visual Studio Team Services control panel on the **Agent queues** tab as shown in Figure 5-7.

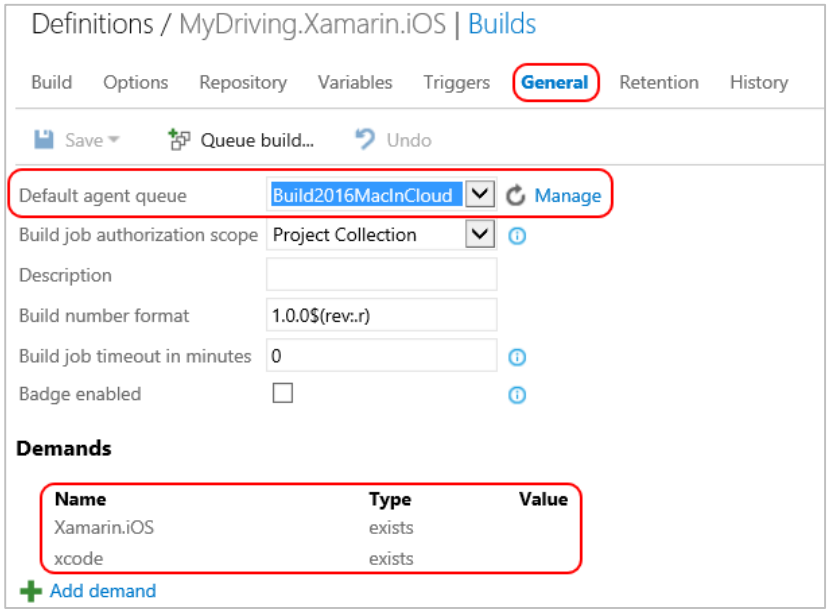


Figure 5-6: Setting a build agent queue in an iOS build definition. Notice the Demands section at the bottom that specifies machine requirements.

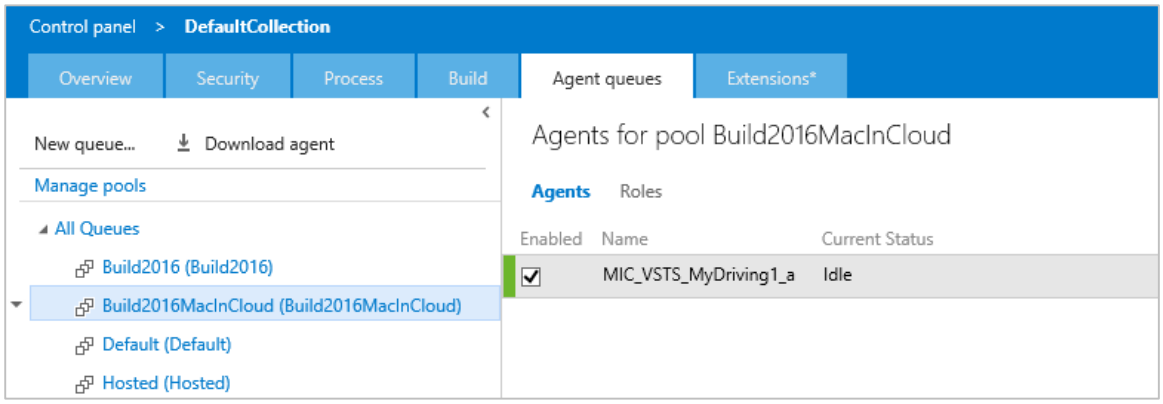


Figure 5-7: Managing build agents in Visual Studio Team Services where MacinCloud machines appear. MacinCloud machines appear here automatically.

Instrumenting the app for telemetry with HockeyApp

As noted before, HockeyApp handles distribution to testers along with telemetry and crash reports.

When you work with HockeyApp, you start by registering your app as described on How to create a new app (hockeyapp.net). In the code, you then include the necessary libraries like HockeySDK.Xamarin (iOS and Android) and HockeySDK.UWP (Windows) as noted in the "Xamarin Components" section of Chapter 3.

In the app code, you make sure to turn on features such as crash reporting and update notifications (shown here for the MyDriving.Android project in Activities/MainActivity.cs):

```
//Member of the MainActivity class, called from MainActivity.OnCreate
void InitializeHockeyApp()
{
    //Logger.HockeyAppAndroid is the app registration key
    HockeyApp.CrashManager.Register(this, Logger.HockeyAppAndroid);
    HockeyApp.UpdateManager.Register(this, Logger.HockeyAppAndroid);
    HockeyApp.Metrics.MetricsManager.Register(this, Application, Logger.HockeyAppAndroid);
    HockeyApp.TraceWriter.Initialize();

    AndroidEnvironment.UnhandledExceptionRaiser += (sender, args) =>
        {
            HockeyApp.TraceWriter.WriteTrace(args.Exception);
            args.Handled = true;
        };
    AppDomain.CurrentDomain.UnhandledException += (sender, args) =>
        HockeyApp.TraceWriter.WriteTrace(args.ExceptionObject);
    TaskScheduler.UnobservedTaskException += (sender, args) =>
        HockeyApp.TraceWriter.WriteTrace(args.Exception);
}
```

The `MainApplication.OnActivityStarted` and `OnActivityStopped` handlers (see MainApplication.cs) also tells HockeyApp to start and stop gathering automatic usage telemetry:

```
public void OnActivityStarted(Activity activity)
{
    CrossCurrentActivity.Current.Activity = activity;
    HockeyApp.Tracking.StartUsage(activity);
}

public void OnActivityStopped(Activity activity)
{
    HockeyApp.Tracking.StopUsage(activity);
}
```

Similar steps are taken in the iOS and Windows app (see AppDelegate.cs and App.xaml.cs, respectively). For complete details on HockeyApp initialization and usage, refer to the documentation for Android, iOS, and Windows.

For user feedback, the app implements a handler for accelerometer shake, which invokes the `HockeyApp.FeedbackManager` API (shown here from the Android project in Activities/BaseActivity.cs):

```
public abstract class BaseActivity : AppCompatActivity, IAccelerometerListener
{
    // [...]
    bool canShowFeedback;

    public void OnShake(float force)
    {
        if (!canShowFeedback)
            return;

        canShowFeedback = false;
        HockeyApp.FeedbackManager.ShowFeedbackActivity(this);
    }
}
```

Deploying to testers via HockeyApp

Although you can accomplish a great deal with unit testing and automated UI testing, some of the most effective testing is done in the field by putting the app into the hands of real people. Therefore we first release any new feature to a small number of friendly testers and then to a larger group. Once we're satisfied with the quality of the app, we're ready to release it to the public app stores. At each stage, of course, we also take feedback and update our designs and priorities accordingly.

HockeyApp manages two parts of the cycle: managing testing on growing sets of test devices, and feedback. There are two types of feedback: words from the users themselves, augmented with screen-shots, and automated reports of crashes and usage counts. In the previous section we already saw how to activate these features in the app.

With the HockeyApp plug-in for Visual Studio Team Services, it's a simple matter to send the built app package to HockeyApp for distribution to testers. First, the build definition's release trigger is set for Continuous Deployment as shown in Figure 5-8. We then configure the HockeyApp plug-in to send the built package to HockeyApp as shown in Figure 5-9 (next page). Once that deployment happens after a build, we can see the release history on the HockeyApp dashboard in Figure 5-10 (next page).

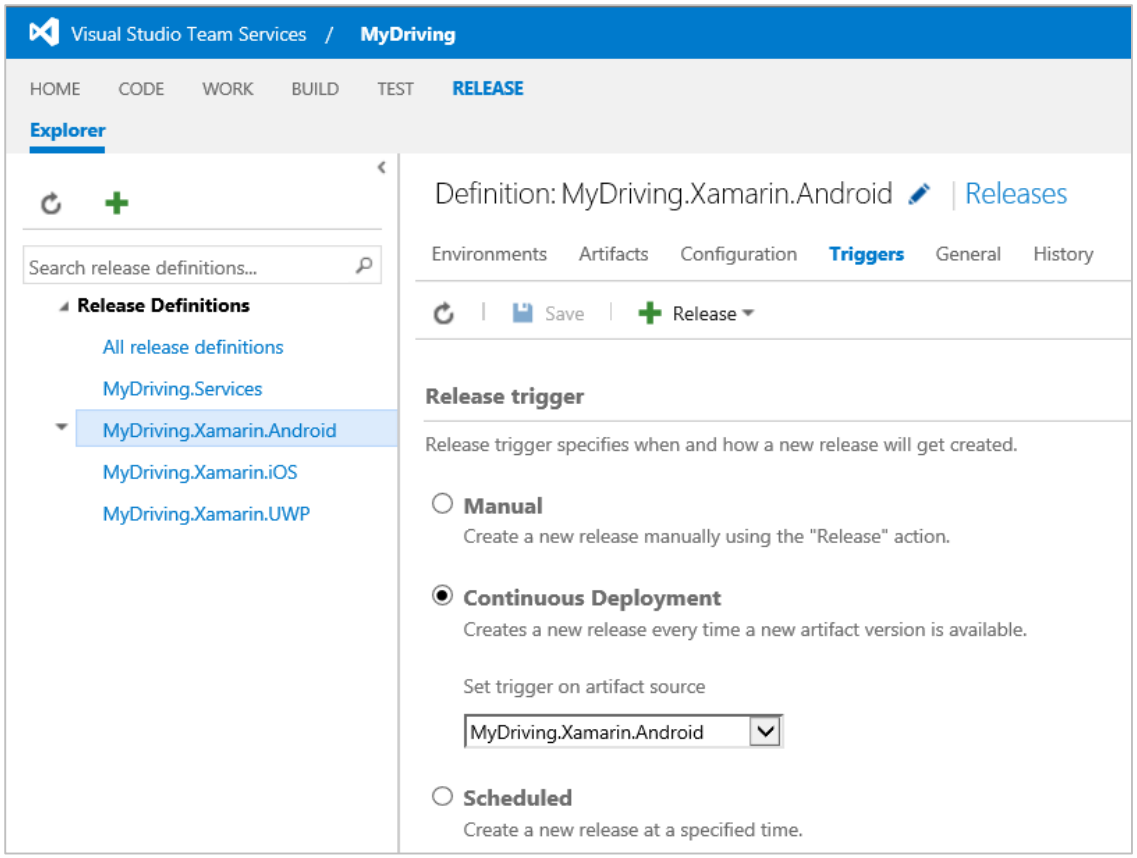


Figure 5-8: Setting a continuous deployment release trigger in Visual Studio Team Services

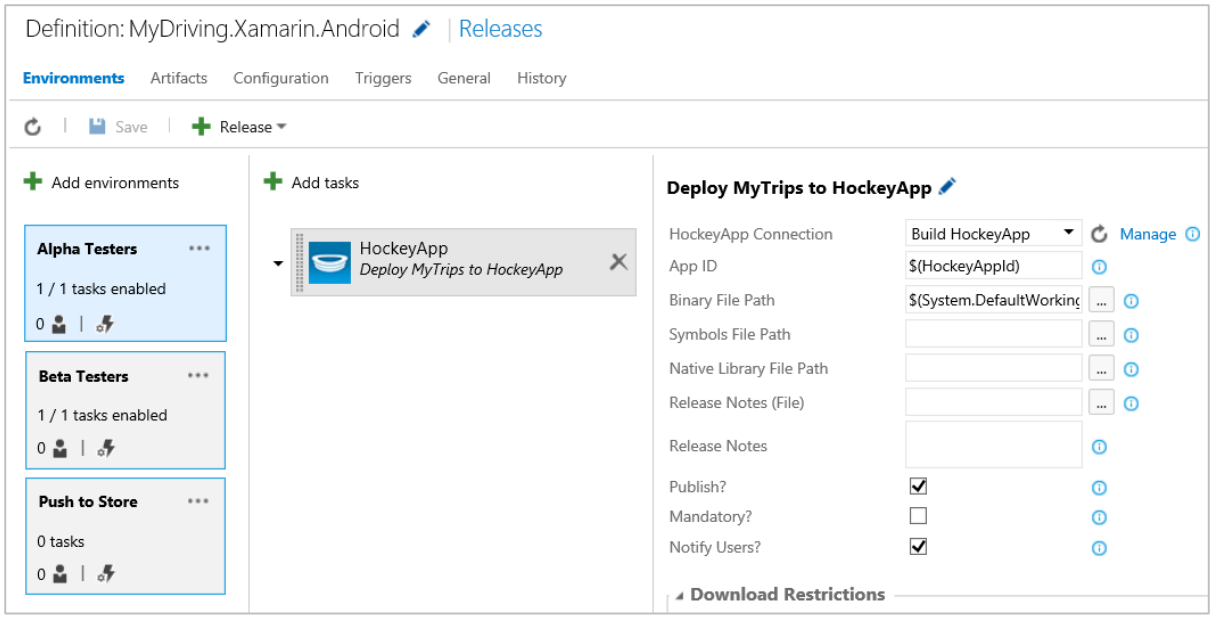


Figure 5-9: Configuring the HockeyApp plug-in to send the built package to HockeyApp

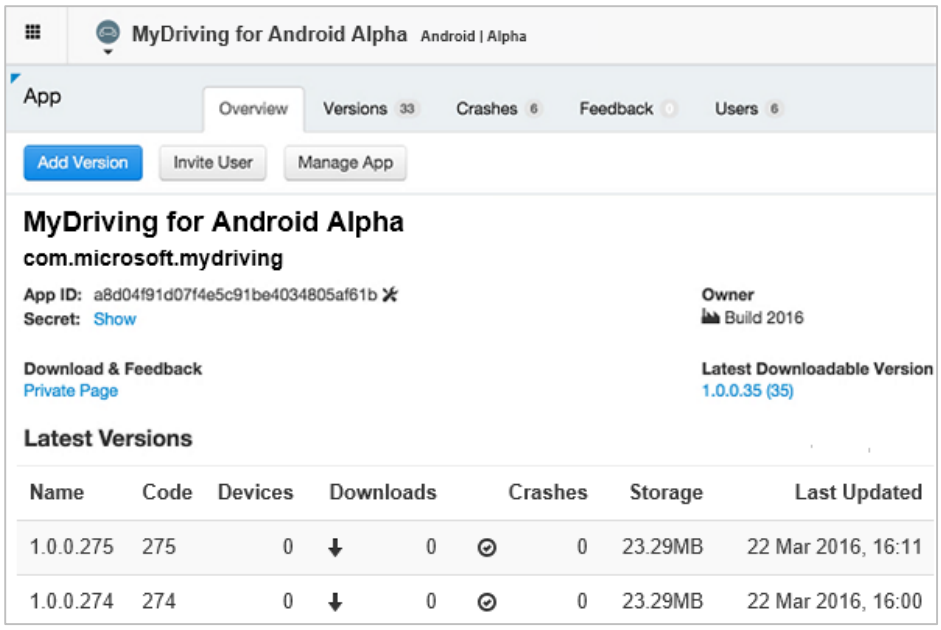


Figure 5-10: Release history on the HockeyApp dashboard

On the HockeyApp dashboard you can also do the following:

- Manage test users, and invite new testers to try your app.
- Monitor feedback from test users.
- Monitor and analyze crashes both from beta releases (with test users) and public releases.

A tester will register his or her devices on their view of the portal, and they'll see the app appear in their private app gallery as shown in Figure 5-11 on the next page. Testers who install the app can then provide feedback through the app or through the portal.

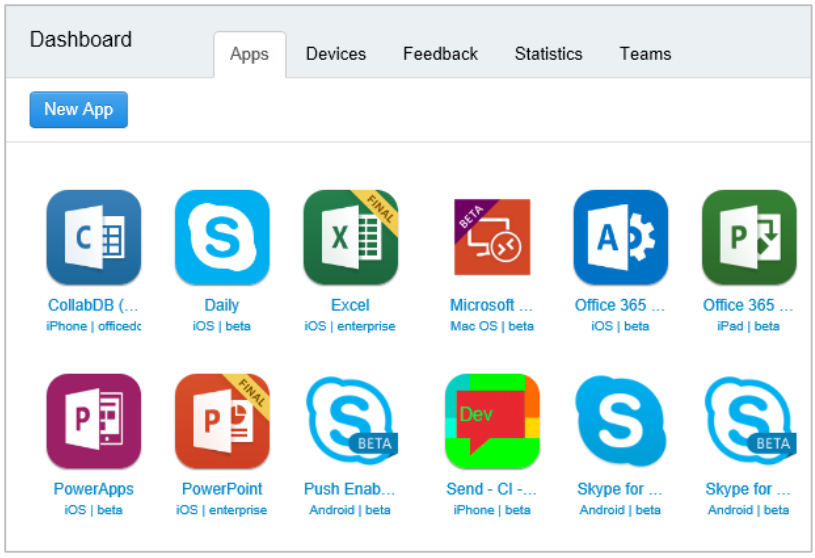


Figure 5-11: Tester view of the HockeyApp portal, showing available apps

Monitoring through HockeyApp

Finally, the HockeyApp dashboard provides the app owner with a summary of crashes and user feedback, including messages from users and automatic screenshots. It groups the crash reports on all platforms by similarities as shown in Figure 5-12, so you can see the critical parts quickly and easily. A detailed log of any crash is available (Figure 5-13, next page). And as you can see in Figure 5-14 (next page), crashes also automatically create a work item in your Visual Studio Team Services backlog.

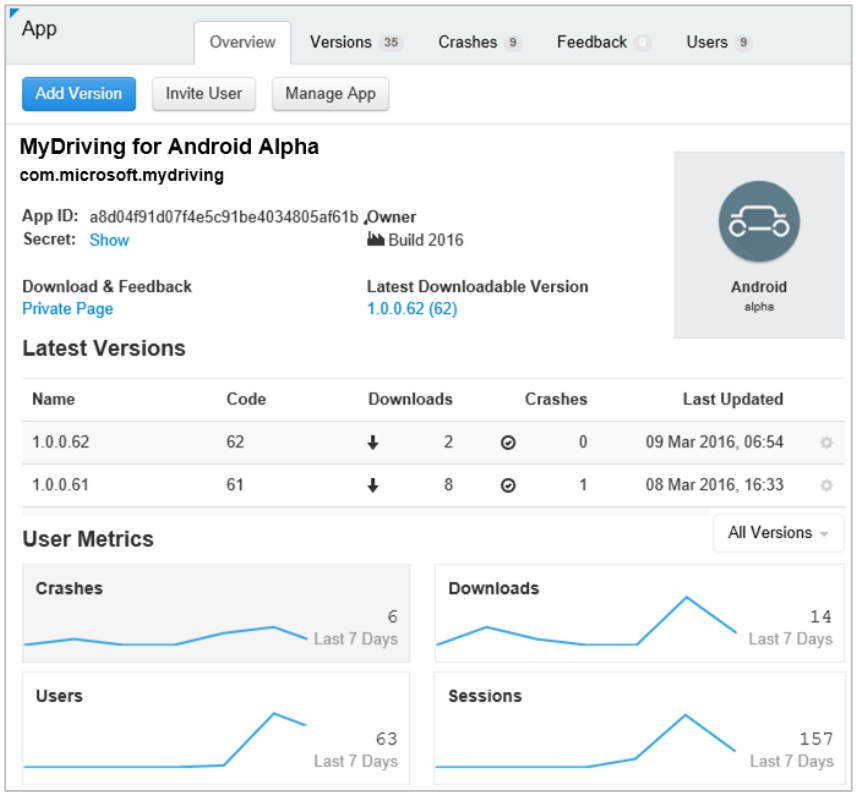


Figure 5-12: Grouped crash reports on the HockeyApp portal

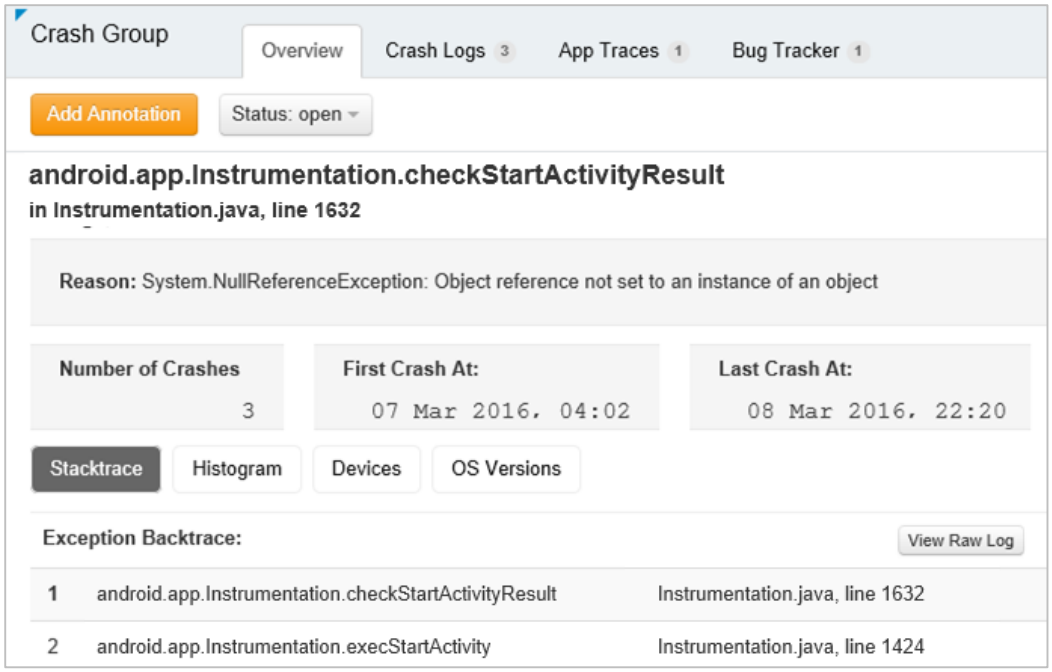


Figure 5-13: A detailed crash report in HockeyApp

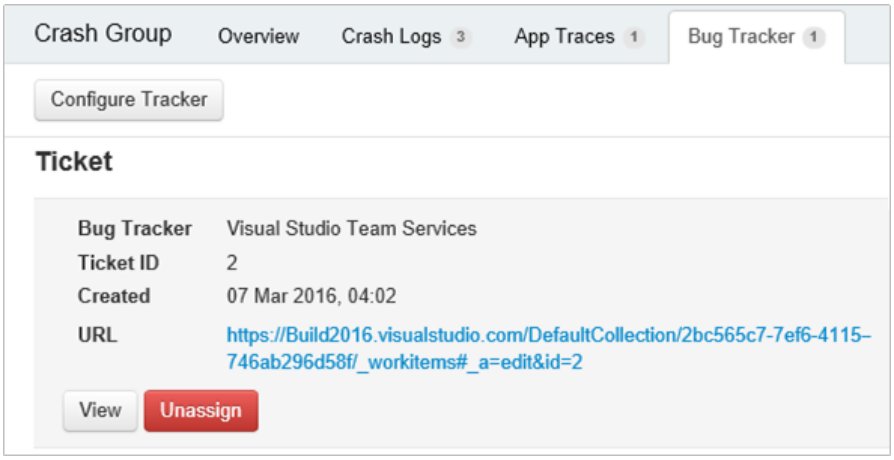


Figure 5-14: Crashes automatically create a work item in your Visual Studio Team Services backlog.

If you've turned on usage tracking in the app, you'll also see usage charts for different features (Figure 5-15, next page). This helps you see test coverage, segmented by device type and OS version. You also get charts of which testers used the app, and for how long, so that you can easily follow up with particular testers.

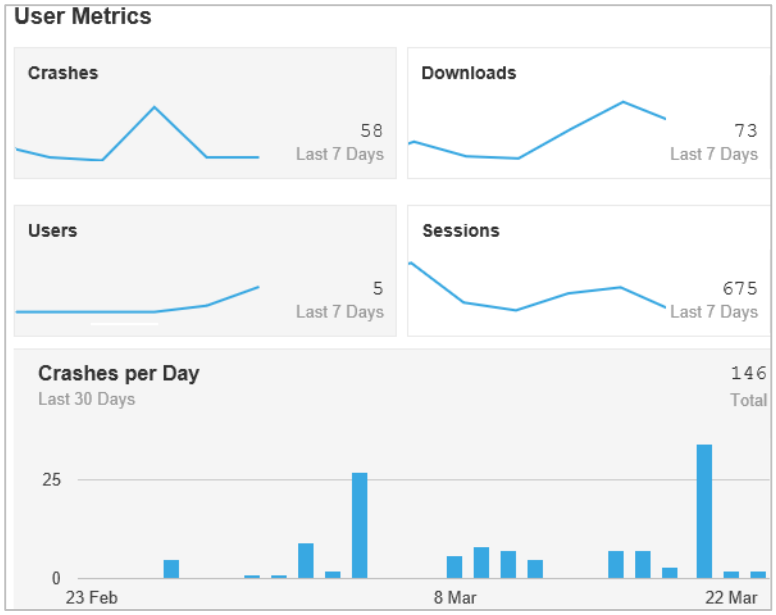


Figure 5-15: Overview charts in HockeyApp

Back end DevOps

The MyDriving back end, as we've seen, is built from many different Azure components. The piece that concerns us for DevOps is the App Service API project (see Chapter 4). As shown in Figure 5-16, it has a similar lifecycle to the mobile app, in that commits to the repository trigger continuous delivery (a build, test, and publish cycle) in Visual Studio Team Services. A successful build is then deployed to App Services, where we use Application Insights to monitor the performance and usage of the live application. Application Insights also provides powerful diagnostic capabilities for issues that arise.

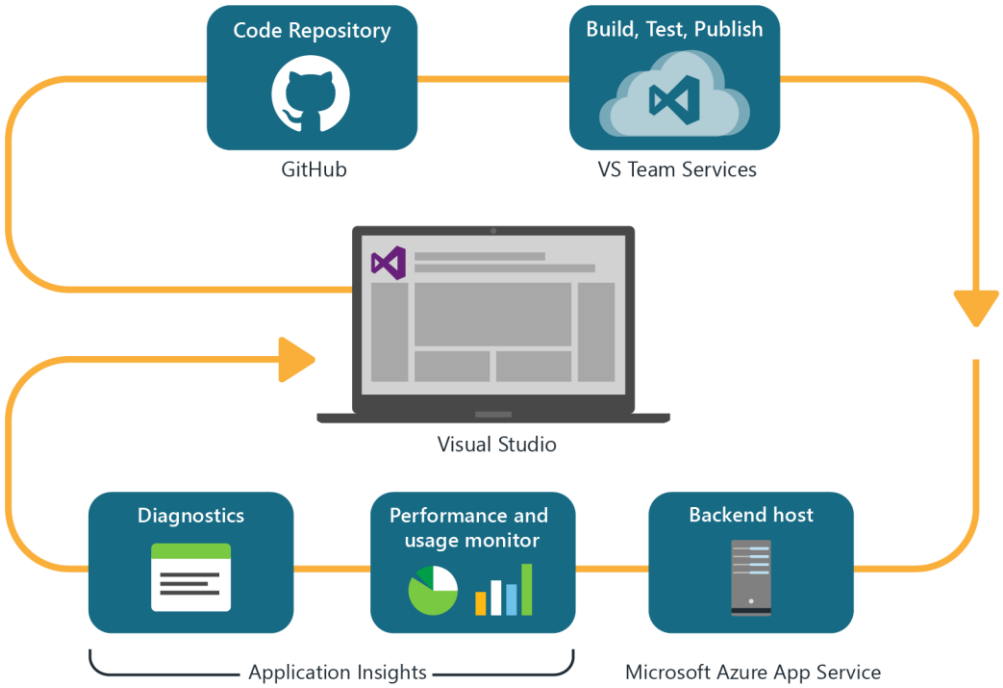


Figure 5-16: Service DevOps with Application Insights

Visual Studio Team Services build definition

The App Service API project is somewhat simpler to build than a Xamarin app. All that's needed is a simple build definition that includes any unit tests (Figure 5-17). The build definition also sets the release trigger for continuous delivery, which sends the successful build to App Service by using the same kind of publishing steps that we saw from Visual Studio in Chapter 4.

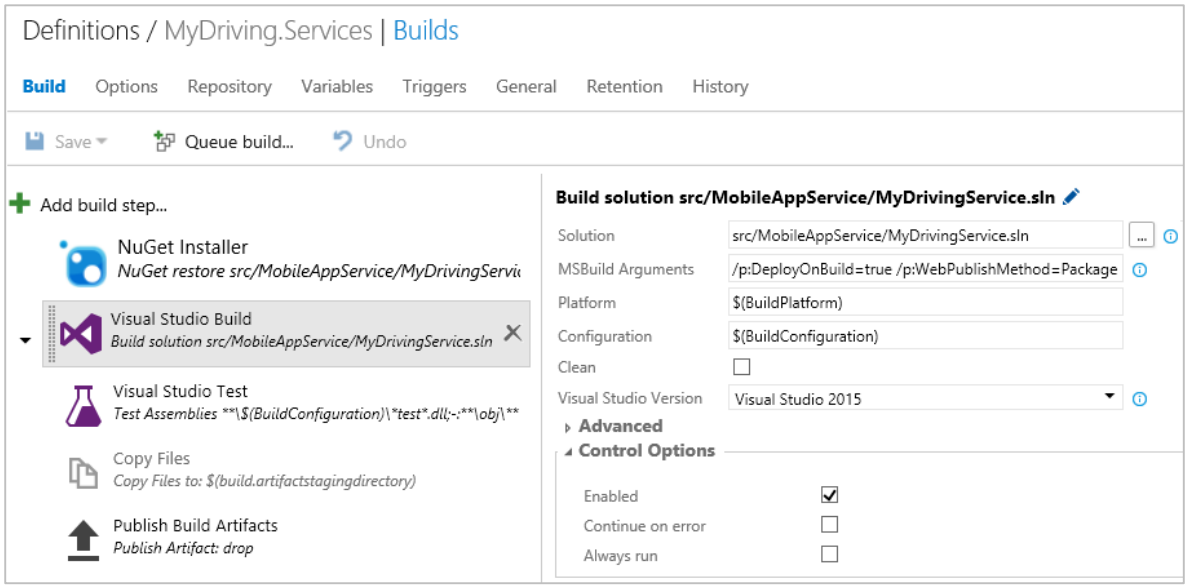


Figure 5-17: Build definition for the MobileAppService project

Adding Application Insights to the API project

By default, App Service automatically provides some performance data such as network, disk, and CPU usage. These help us decide if we need to scale up and ask Azure to assign us more resources. To get more detailed information, we turn to Application Insights.

Adding Application Insights to the App Service API project in Visual Studio is a simple matter that's described on Set up Application Insights for ASP.NET. Just right-click the project and select **Add Application Insights Telemetry...** as shown in Figure 5-18.

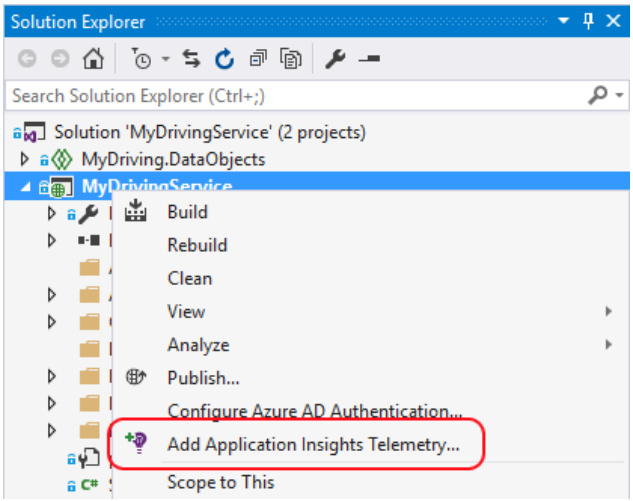


Figure 5-18: One click adds Application Insights to a service project

This command starts a UI through whch you can adds the necessary SDK to your project and sets up a space for your app in the Application Insights service. You need a subscription to Microsoft Azure, of course, but Application Insights doesn't cost anything for low volumes of telemetry.

The SDK sends telemetry about the performance of your app to the Application Insights service, where you can monitor performance and diagnose any problems. During debugging, you can also see and search the diagnostic logs in Visual Studio (Figure 5-19).

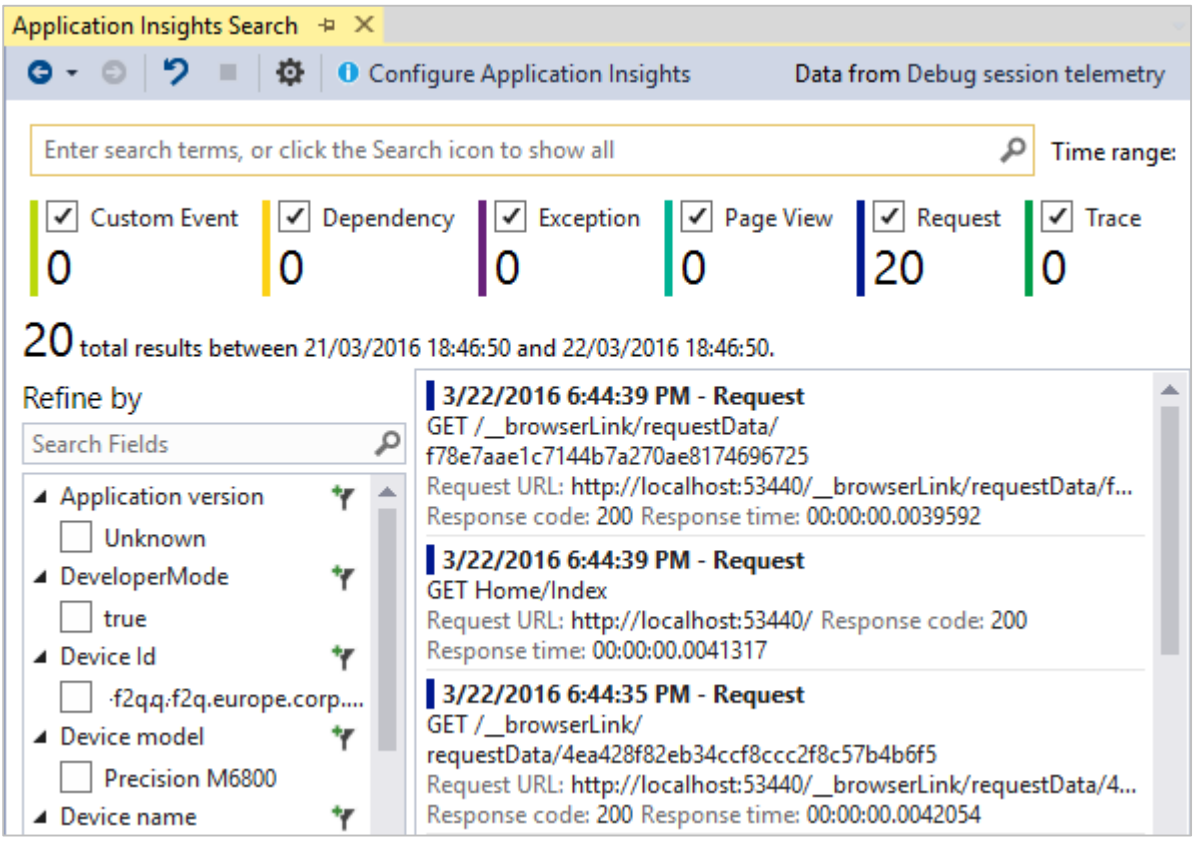


Figure 5-19: Exploring Application Insights telemetry within Visual Studio

By default, the SDK logs data for the following:

- The rate of HTTP requests and the speed and result codes of the app's responses to them.
- The rate, success, and response times of SQL operations and REST calls to other components of the application.
- Exceptions.
- Performance metrics such as CPU and disc usage.

In addition, you can write trace calls to track any other metrics and events that you're interested in monitoring.

Note The MobileAppService project in MyDriving doesn't actually include the Application Insights SDK directly because it doesn't need any custom features. Application Insights is instead configured directly in Azure for the service by means of the ARM template.

Monitoring through Application Insights

Application Insights monitors the performance and usage of the app. It provides valuable diagnostic search tools if any problems occur, and sends alerts if there are sudden suspicious rises in exceptions or failed request rates as shown in Figure 5-20.

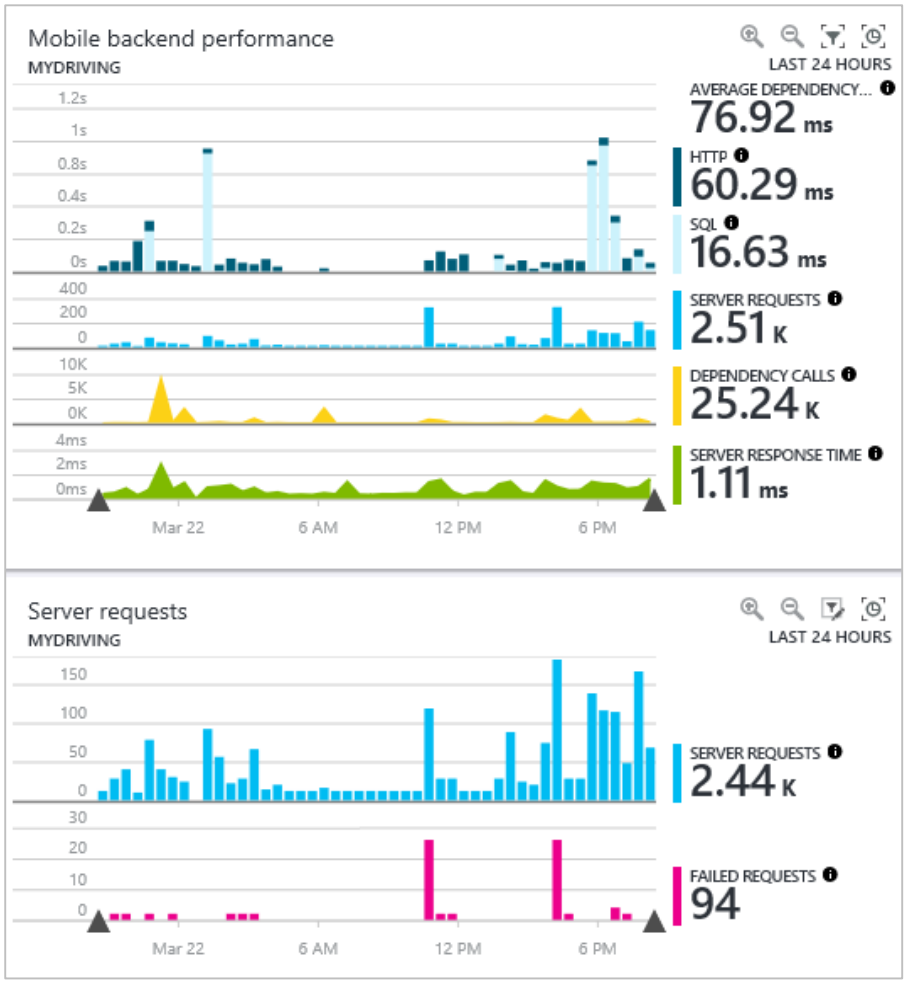


Figure 5-20: Part of an Application Insights dashboard

All this telemetry is sent to the Application Insights service, which creates performance and usage dashboards and provides diagnostic tools. When you install the SDK in the project, the process also sets up an Application Insights resource in Azure, which you use to look at the charts or do diagnostic searches when the app is live (Figure 5-21).

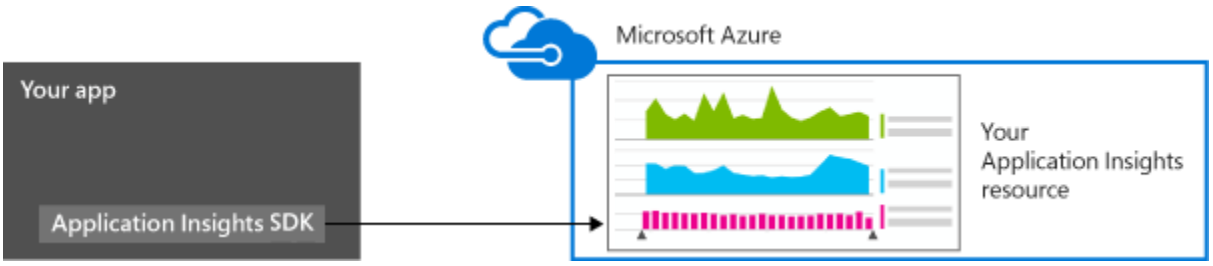


Figure 5-21: Azure shows a resource for Application Insights for App Service projects that use the Application Insights SDK.

Here's what you're able to do on the dashboard:

Performance tuning: One of the key charts in Application Insights shows response times together with request counts over the past day (or week, or hour, and so on, as shown in Figure 5-22). This chart shows usage as well as performance under load. If there's a sudden rise in the response time as request rates goes over a particular number, then you'll know you've hit a bottleneck somewhere. It might be as simple as not having enough CPU power; or it might be that one of your dependencies is letting you down.

There's a set of charts that shows the rates and response times for calls from your application to external dependencies such as databases or REST APIs. If the rise in your application's response time is accompanied by a rise in response times to calls to another component, then you know that your problem is with that dependency—not inside your app.

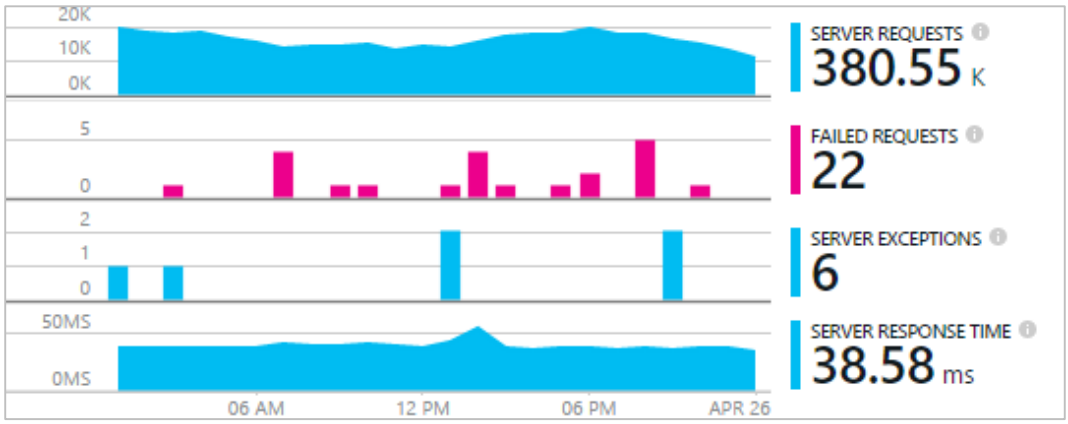


Figure 5-22: Charts of response times and requests in Application Insights

Alerts: Application Insights automatically sends you email alerts if your app is responding slowly to HTTP requests. (You can also manually set up alerts to be sent when various metrics cross a threshold.) This helps you respond quickly if there's a sudden surge of interest in your app—or if your most recent release turns out to have some issues.

You can also get alerts on rates of failed requests and on exceptions. Every busy app has a few of these, so you won't get an email for every individual exception (unless you really want that); but Application Insights looks at the rate of these occurrences and alerts you if the rate rises.

Finally, Application Insights also has web tests (Figure 5-23) to ping the service from around the globe, and to check that it's still live and visible from the outside world.

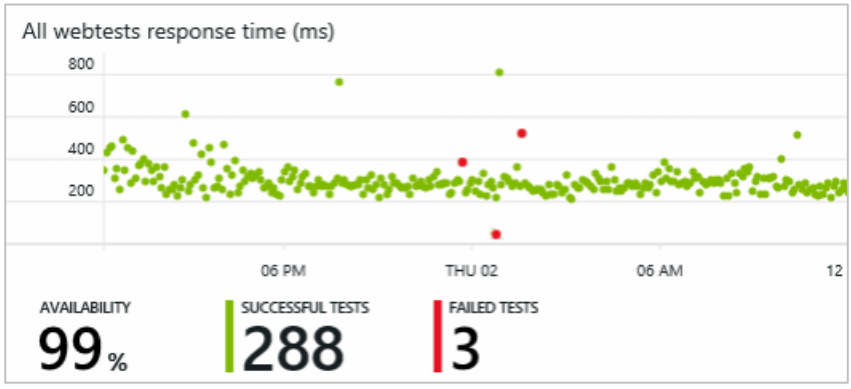


Figure 5-23: Application Insights graph of web tests

Diagnostics: When you do have a problem, the first thing you want to do is look at specific events and find out more about them. Application Insights comes with some particularly powerful search tools (Figure 5-24) that let you make SQL-like queries over the logged requests, exceptions, and other events. For example, there's a pattern-finding query that can find out if failed requests had any particular combination of properties, such as a particular client type or location. Particularly useful are the dependency traces, which show issues that originate in one of the other components of the application.

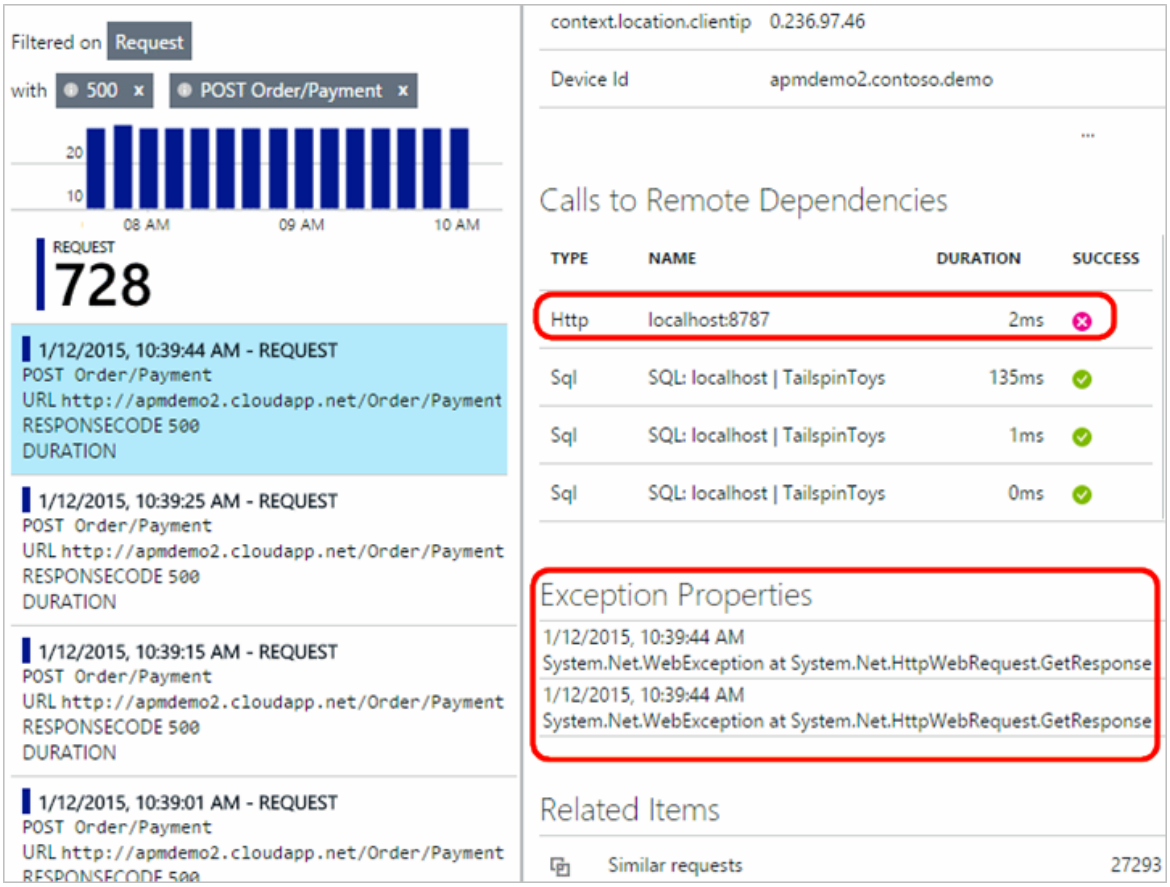


Figure 5-24: Search blade of Application Insights

In addition, there's a powerful analytical search capability with which you can perform SQL-like queries over the telemetry, shown in Figure 5-25 (next page). You can filter, join, aggregate, derive calculated values, and get statistical charts. It's great both for pinpointing specific trouble spots and for answering questions about performance and usage.

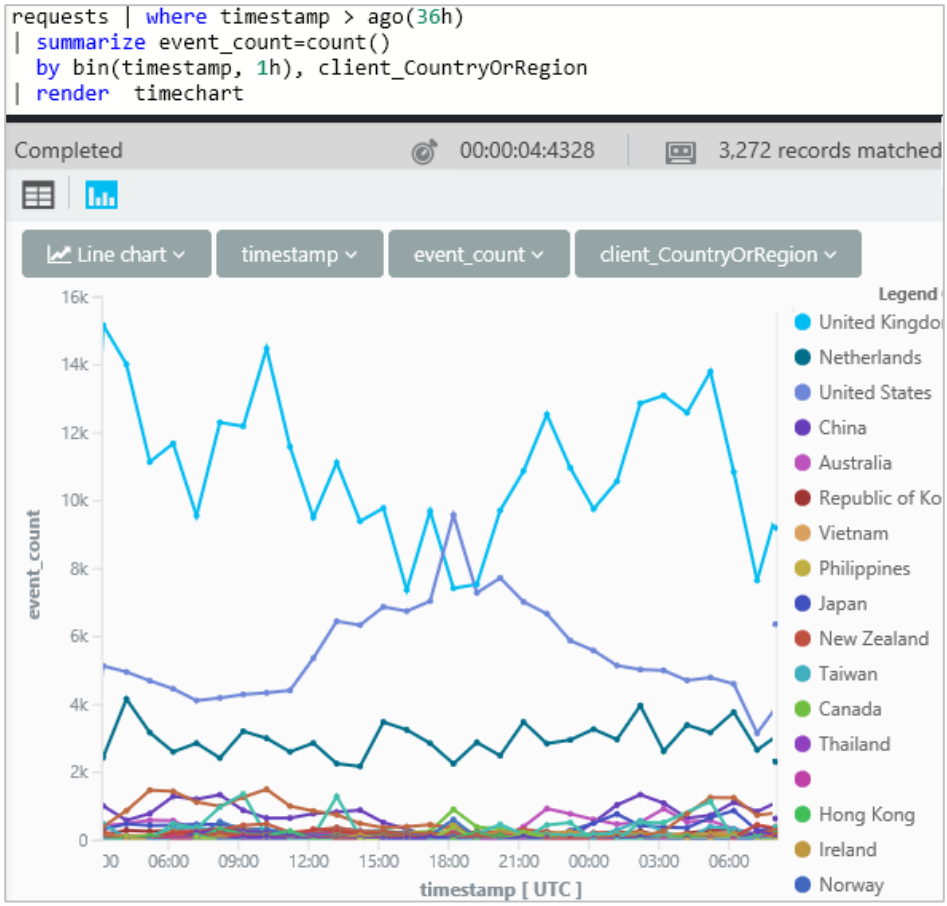


Figure 5-25: Analytics in Application Insights

Usage: We'd like to know what our users are doing with the app. HockeyApp gives us a lot of usage data straight from the devices. The main purpose there is to show how much each feature has been exercised. But we can also instrument measurements in the server code, and send them as telemetry to Application Insights. In MyDriving, for example, we can send an event whenever a user completes a journey, and include driving score and length of journey along with the event. We can track the numbers of users achieving different levels of competence, and segment and chart those on the dashboard. Alternately, as we'll see in the next chapter, we can also derive such insights straight from the IoT data stream and visualize them in Power BI.

Display: Application Insights has its own good dashboard and charting features, but it can also output to Power BI or other tools. This capability lets us combine the server analytics with monitoring data from HockeyApp from IoT ingestion.

Real-Time Data Handling

Having seen an overview of the MyDriving system, the nature of IoT devices, the role of the mobile app, the App Service API endpoints, and our DevOps setup, we're now ready to explore what's happening internally on the back-end. We'll do this by following distinct flows of data through the system: real-time data (this chapter), aggregate data through machine learning (Chapter 7), aggregate historical data (Chapter 8), and real-time data through microservice extensions (Chapter 9).

In the real time or "hot path" data flow, IoT data is processed by the Azure back end as it arrives. Results are made available for visualization via the mobile app and Power BI as shown in Figure 6-1.

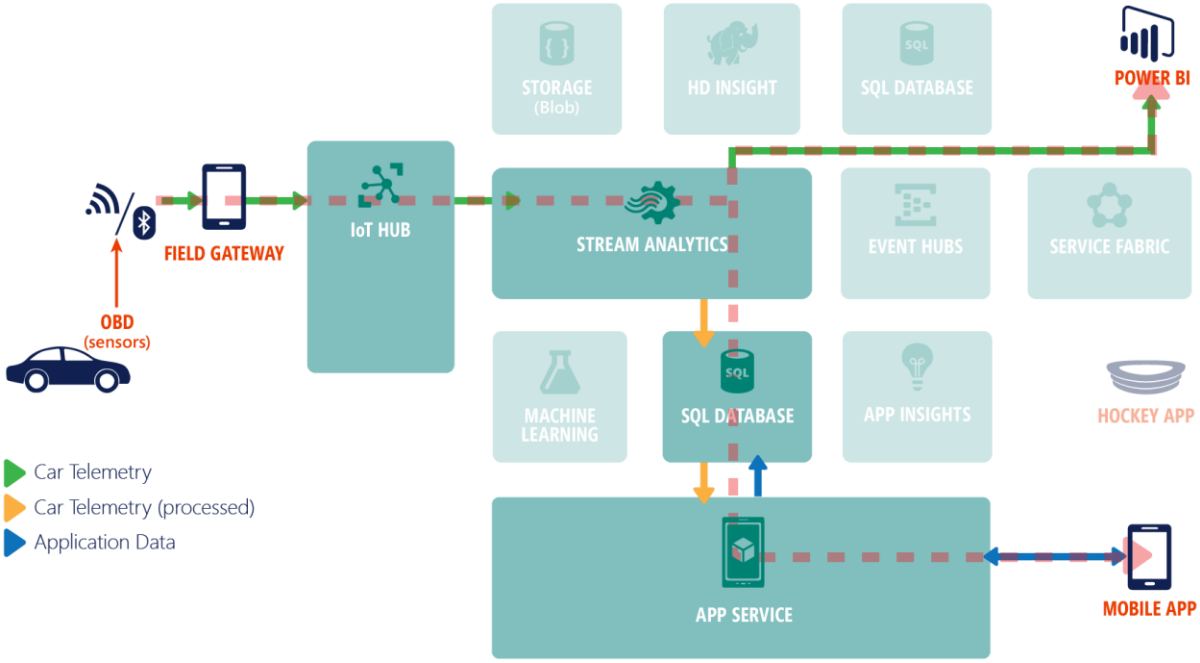


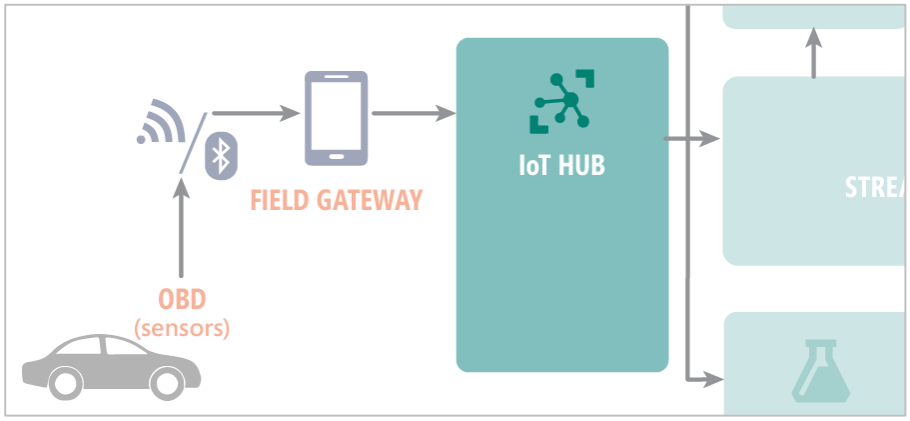
Figure 6-1: The real-time or "hot path" data flow in MyDriving

Data enters the back end through IoT Hub, where it's picked up by an always-running Stream Analytics job named `mydriving-sqlpbi`. This job applies a query to shape the data, then feeds it into two outputs. The first output, named `SQLSink`, is the SQL database that's backing the App Service table storage exposed through REST APIs. The second output, `PowerBISink`, is linked to a DataSet and Table in a connected Power BI subscription, named `MyDriving-ASAdataset` and `TripPointData`, respectively.

The specific shaping that happens here, as we'll see in the query a little later on, is to compute ongoing averages, minimums, and maximums for each data set coming from IoT Hub. This really says that the data of interest on the consumption side of the system is concerned with these sorts of aggregate values, and not the lower-level details. In your own deployments, of course, you can have the Stream Analytics job produce different results simply by changing the query, thereby making those results available to other parts of the system.

There are clearly a number of pieces we now need to dive into further. We'll start with IoT Hub, followed by Stream Analytics and storage, and wrap up with Power BI. As we've done earlier in this documentation, we'll provide a short primer for each of the services in question, and then see their roles and relationships within in the MyDriving architecture. Because we're covering a number of different services here, we include a small graphic at the beginning of each section to give you a visual reminder of where the service sits in the overall MyDriving architecture.

IoT Hub



In MyDriving, IoT Hub is the starting point for the flow of vehicle telemetry data through the back end.

Primer: IoT Hub

Azure IoT Hub is a fully-managed service that enables reliable and secure bi-directional communications between millions of IoT devices and a solution back end. Its main characteristics are as follows:

- **Provides reliable device-to-cloud and cloud-to-device messaging at scale.** IoT Hub can reliably receive millions of messages per second from millions of devices. It also retains the message data for up to seven days to guarantee reliable processing by the back end, and to absorb peaks in the load. The back end can use IoT Hub to send messages with an at-least-once delivery guarantee to individual devices. For more information, see the "Messaging" section of the Azure IoT Hub developer guide.
- **Enables secure communications using per-device security credentials and access control.** You can provision each device with its own security key to enable it to connect to IoT Hub.

The IoT hub *identity registry* stores the device identities and keys for your solution. A custom back end can allow or deny individual devices by enabling or disabling them in the identity registry. For more information, see the "Device identity registry" section of the Azure IoT Hub developer guide.

- **Provides monitoring for device connectivity and device identity management events.** You can receive detailed operation logs about device identity management operations and device connectivity events. This enables your IoT solution to easily identify connectivity issues, such as devices that try to connect with wrong credentials, send messages too frequently, or reject all cloud-to-device messages sent to them. For more information, see Introduction to operations monitoring and Introduction to diagnostic metrics.
- **Includes device libraries for the most popular languages and platforms.** Azure IoT device SDKs are available and supported for a variety of languages and platforms. There is a C library (designed specifically to simplify porting across platforms) for many Linux distributions, Windows, and real-time operating systems. There are also SDKs for managed languages, such as C#, Java, and JavaScript. The MyDriving phone app uses the PCL version of the .NET library that uses the HTTP protocol by default.

IoT Hub exposes several endpoints that enable devices and other cloud-based services to interact with the service as shown in Figure 6-2.

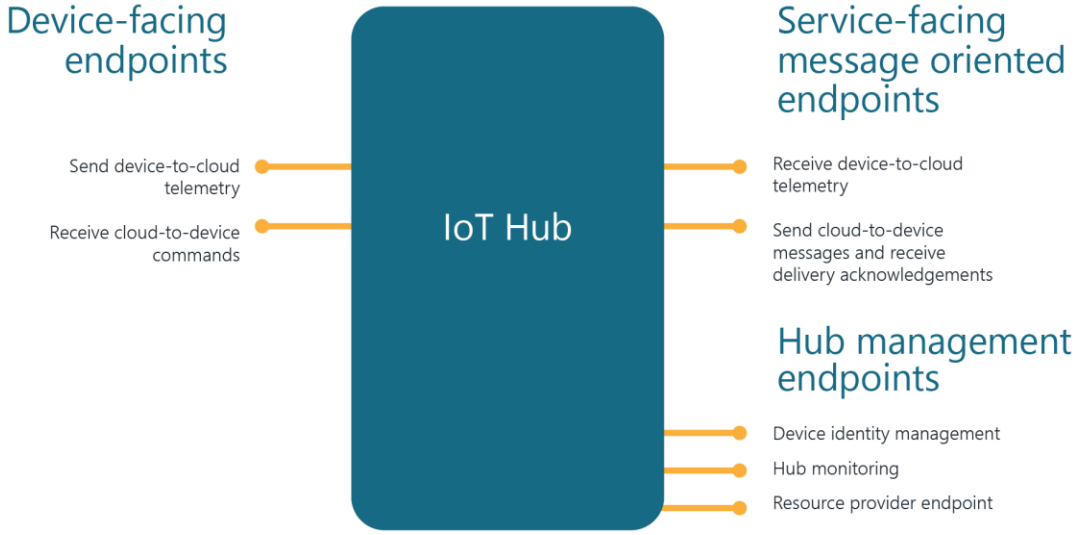


Figure 6-2: Endpoints exposed by IoT Hub

Each IoT hub exposes two device-facing endpoints, accessed using HTTP, MQTT, or AMQPS:

- **Send device-to-cloud telemetry.** A device can connect to this endpoint to send messages to the back end. In MyDriving, the mobile app (acting as a field gateway) connects to this endpoint over HTTP to transmit OBD data.
- **Receive cloud-to-device commands.** A device can connect to this endpoint to retrieve any messages sent to the device by the back end. IoT Hub maintains a separate queue for each device to deliver these cloud-to-device messages. MyDriving doesn't use this endpoint.

Each IoT hub exposes three service endpoints you can access using the AMQPS protocol:

- **Receive device-to-cloud telemetry.** A back-end service can connect to this endpoint to retrieve any messages sent to the hub from a device. This endpoint is compatible with Event Hubs (see Chapter 9, *Microservice Extensions*) and you can connect to it using any of the

interfaces and SDKs that support Event Hubs.

In MyDriving, the four Stream Analytics jobs connect directly to this endpoint to retrieve telemetry sent from the car. The MyDriving IoT Hub is configured with four consumer groups that enable each of the Stream Analytics jobs to read the car telemetry at their own pace.

- **Send cloud-to-device commands and receive delivery acknowledgments.** These two endpoints enable your back end to send reliable cloud-to-device messages, and to receive the corresponding delivery or expiration acknowledgments. MyDriving doesn't use these.

In addition to the message oriented endpoints, IoT Hub also exposes the following:

- **Device identity management.** Use this HTTP endpoint to manage the device identity registry in IoT hub. You can create, retrieve, update, and delete the device identities that the solution uses for device authentication and access control. The provisioning API in the MyDriving backend's App Service that you saw in Chapter 4, *App Service API Endpoints,* uses this endpoint through the SDK's `RegistryManager` object.
- **Hub monitoring.** Use this Event Hub-compatible endpoint to collect monitoring data from your IoT hub about device identity operations, device-to-cloud communications, cloud-to-device communications, and connections to the hub.
- **Resource provider.** Use this Azure Resource Manager interface to enable Azure subscription owners to create IoT hubs, update hub properties, delete hubs, and perform bulk device identity import and export operations. The Resource Manager template that deploys MyDriving to your own Azure subscription makes use of this endpoint.

IoT Hub security and the service-assisted communication pattern

IoT Hub works well with the service-assisted communication pattern, which describes how to mediate the interactions between your IoT devices and your back end. The goal of service-assisted communication is to establish trustworthy, bidirectional communication paths between a control system, such as IoT Hub, and special-purpose devices that are deployed in untrusted physical space. The pattern establishes the following principles:

- Security takes precedence over all other capabilities.
- Devices do not accept unsolicited network information. A device establishes all connections and routes in an outbound-only fashion. For a device to receive a command from the back end, the device must regularly initiate a connection to check for any pending commands to process.
- Devices should only connect to or establish routes to well-known services they are peered with, such as IoT Hub.
- The communication path between device and service or between device and gateway is secured at the application protocol layer.
- System-level authorization and authentication are based on per-device identities. They make access credentials and permissions nearly instantly revocable.
- Bidirectional communication for devices that connect sporadically due to power or connectivity concerns is facilitated by holding commands and device notifications until a device connects to receive them. IoT Hub maintains device-specific queues for the commands it sends. This might introduce latency to command handling, because the device doesn't act on the command until the device connects to the queue and retrieves the command.

The mobile industry has successfully used the service-assisted communication pattern at enormous scale to implement push notification services such as Windows Push Notification Services, Google Cloud Messaging, and Apple Push Notification Service.

In MyDriving, the phone acts as a field gateway and does not listen on any port for incoming connections from IoT Hub. The phone itself establishes an outbound connection to the device-to-cloud messaging endpoint on the hub to send telemetry.

Decision point: choosing IoT Hub

IoT Hub performs three roles in MyDriving:

- It reliably receives telemetry collected by the OBD devices in multiple vehicles into the cloud infrastructure.
- It makes the telemetry data available to other cloud services for storage and processing.
- It ensures that only known, registered devices can connect to and communicate with the cloud-based infrastructure.

There are two Azure services that enable telemetry ingestion to the cloud at scale: IoT Hub and Event Hubs. The article Comparison of IoT Hub and Event Hubs offers a detailed comparison of these services and their intended uses, but for MyDriving the key reasons for selecting IoT Hub over Event Hubs are as follows:

- **Security**: IoT Hub provides per-device identity and revocable access control. Every device that connects to IoT Hub has a unique identity, and must be registered in the IoT device identity registry before it can connect to the hub. This guarantees that the source of every message that IoT Hub receives is known. It also means that the access granted to a specific device can be revoked at any time. Event Hubs uses shared access policies to grant access, so if your solution requires per-device credentials, you must implement this functionality yourself. Shared access policies also offer only limited revocation support.
- **Operations monitoring**: IoT Hub enables monitoring through a rich set of device identity management and connectivity events, such as individual device authentication errors, throttling, and bad format exceptions. These events enable you to quickly identify connectivity problems at the individual device level. Event Hubs offers only aggregate metrics.
- **Device SDKs**: IoT Hub provides device SDKs for a range of languages and platforms. This makes it easier to build custom OBD devices that can connect directly to IoT Hub, without the need for a field gateway.

Provisioning devices in IoT Hub

Setting up an IoT Hub in Azure is a straightforward process that's described in Manage IoT hubs through the Azure portal. (The MyDriving solution uses a Resource Manager template to set up the IoT hub. In a large solution that uses multiple, related Azure services, it's easier to create and configure them by using a Resource Manager template rather than manually through the portal.)

On the **Settings** blade, you configure aspects like shared access policies and keys, and define the connection strings that are used to connect to the hub. On the **Messaging** blade, you then configure how the hub receives and stores messages, and access the Event Hubs-compatible name and endpoint values that are used to retrieve messages sent from devices. This Event Hubs-compatible endpoint is the one that the Stream Analytics job in MyDriving uses for extensibility, as described in Chapter 9.

Once a hub is set up, provisioning a device requires two steps:

1. Create an entry for the device in the device identity registry in your IoT Hub. Registering a device creates a unique identity in the registry, with an associated key that the device can use to authenticate with the hub. As we saw in Chapter 3, *The Mobile App,* the app includes this key in the connection string it uses to connect to IoT Hub.
2. Install the device key on the device to enable the device to connect to your IoT hub.

At a high level, the device identity registry is a REST-capable collection of device identity resources. From a coding point of view, as you saw in Chapter 3, you typically work with the RegistryManager class in the Azure IoT SDK. It provides easy access to the operations in the hub's registry as show in Figure 6-3.

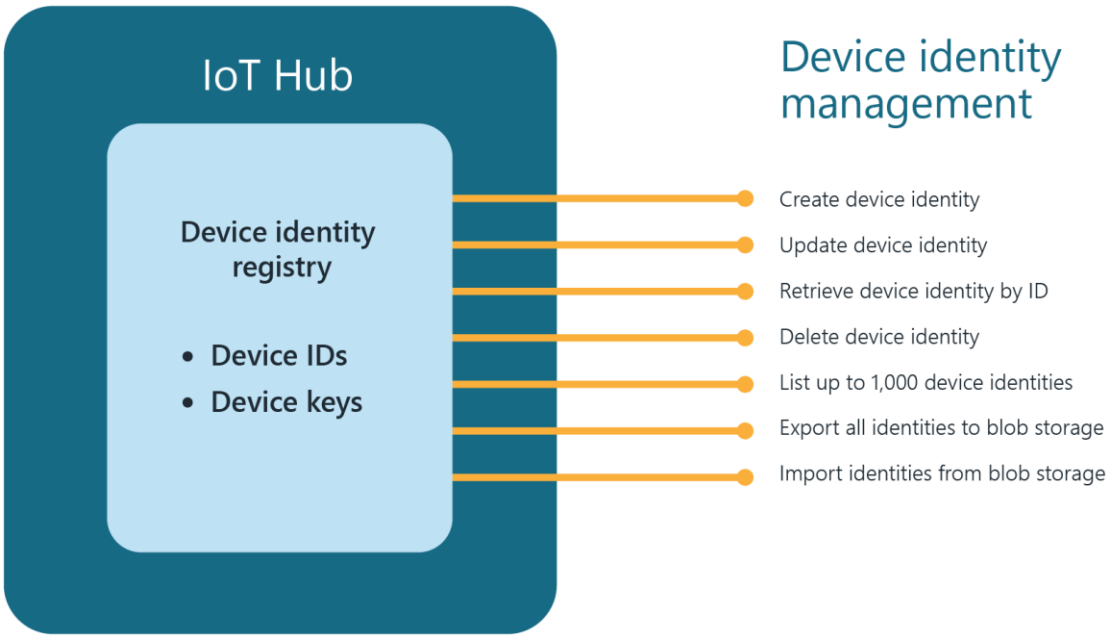


Figure 6-3: IoT Hub registry operations

In MyDriving, the phone acts as an opaque field gateway as described in Chapter 2, *IoT Devices*. Therefore, the IoT hub provisions the app with a key from the device identity registry. The app again uses this key with the hub's connection string to send data. In MyDriving, that data is picked up by a Stream Analytics job as we'll see next. It could also be picked up by using an `EventProcessorHost` instance as described in the How to process IoT Hub device-to-cloud messages tutorial, but that isn't covered here.

Note also that MyDriving doesn't use IoT Hub's ability to send commands from the back end to the devices themselves. For an example of that process, see How to send cloud-to-device messages with IoT Hub.

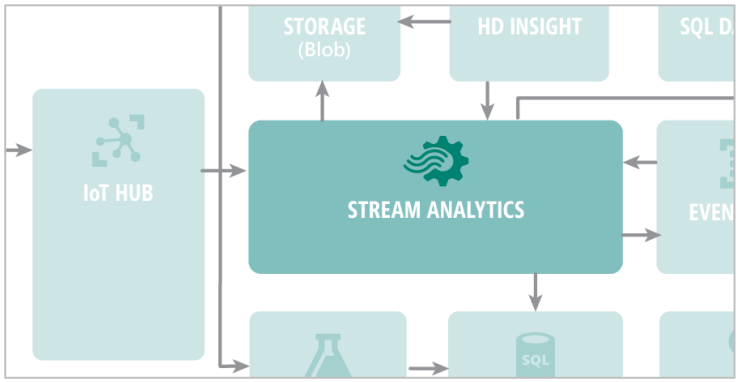
Stream Analytics and storage

In the MyDriving architecture, Stream Analytics processes the data from the IoT devices that it receives through IoT Hub. The task of Stream Analytics is to reshape the data so that it can be acted

on and shared via App Service storage for consumption in the mobile app. The data is then sent directly to Power BI (for monitoring the whole system) and Event Hubs (for extensions).

This section gives an overview of Stream Analytics, along with details of its usage in MyDriving. Along the way we'll also look at Azure storage, which is where much of the data gets stored along the way.

Primer: Stream Analytics



Azure Stream Analytics performs fast processing *jobs* on data streaming from anywhere—devices, sensors, and the web. Typically this includes aggregating and filtering that data, and perhaps also combining it with static data. It's applicable wherever there's a requirement to process a continuous stream of data, such as monitoring machinery or an environment (as in MyDriving), or finding faults and summarizing usage in application telemetry.

A Stream Analytics job uses a SQL-like language to express real-time data transformations over the incoming data as illustrated in Figure 6-4. You can filter, join, aggregate, and transform data. Results are continuously streamed to the sinks you specify. Like other Azure services, it's highly scalable, and helps with security and reliability.

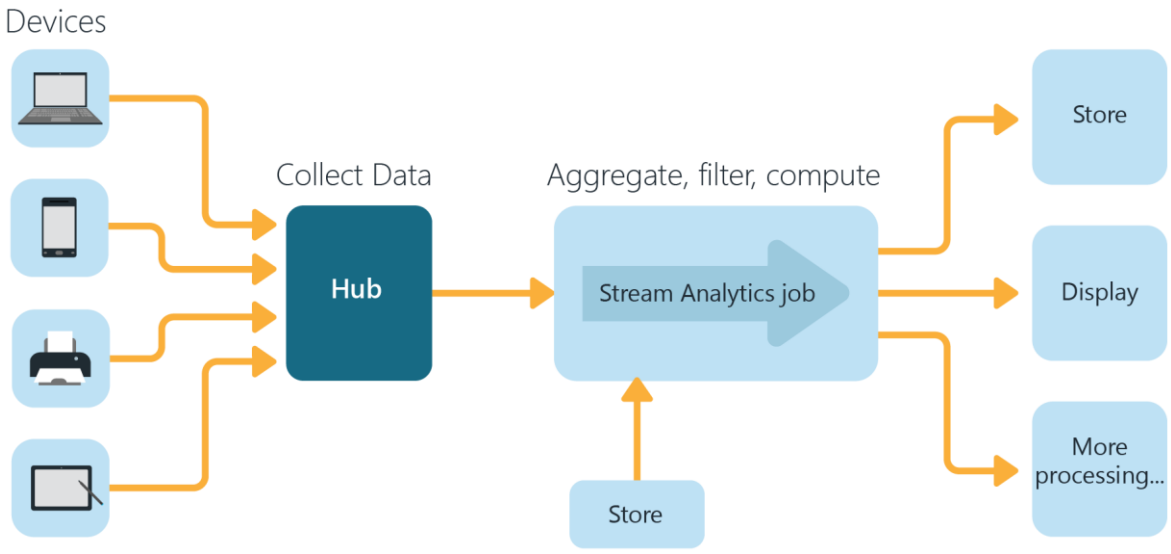


Figure 6-4: The general role of Stream Analytics.

Streaming architectures provide a notable contrast with more traditional architectures. In the more traditional approach, incoming data is immediately loaded into a database, which is then queried by

applications. Updating the database is cumbersome, and it can be minutes before critical queries can be performed on the uploaded data.

In a streaming solution, the critical queries are processed continuously as they arrive, and immediately passed on to a suitable sink. The inputs and outputs that are directly connected to Stream Analytics are typically queues, stores, or databases, with which other services can work at their own rate.

A Stream Analytics job is very simple: it has some *inputs*, a processing *query*, and some *outputs*. A typical Stream Analytics job, such as `mydriving-sqlpbi`, appears in the Azure portal as shown in Figure 6-5.

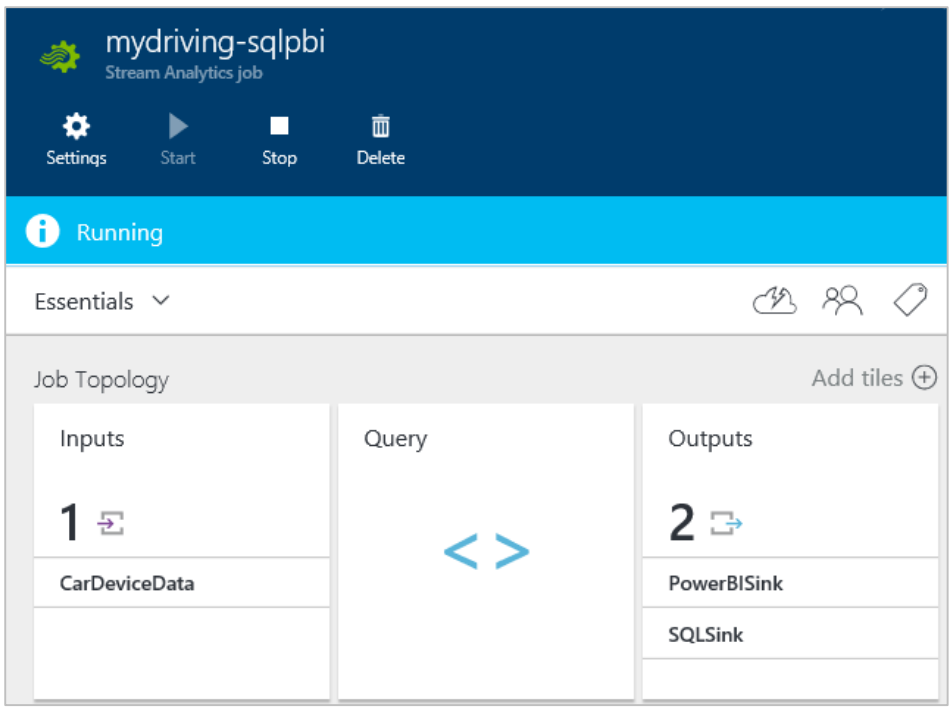


Figure 6-5: The mydriving-sqlpbi job in Azure Stream Analytics

The inputs and outputs are other Azure services. It's easiest to set up those services first, then connect a Stream Analytics job between them. To plug them into Stream Analytics, just add a new input or output definition, and then copy across the relevant keys and IDs.

There are a variety of adapters for coupling to different types of inputs and outputs. Inputs can be sourced from several types of services:

- **IoT Hub**: Gathers data from external devices, with secure, validated two-way connections. In our demo app, that's how we get data from vehicles.
- **Event Hubs**: Can accept data from devices at high volume, though without identifying individual devices. Event hubs have a broad application as high-volume buffers between co-located Azure services. They can be connected both as inputs and outputs to Stream Analytics.
- **Blob storage**: Used for large amounts of unstructured data. As an input, blob storage is typically used to join relatively static or "cold path" data into Stream Analytics queries.

Outputs can sink to:

- **Blob storage**: Used for storing output as a series of timestamped chunks.
- **Document DB**: Used for storing JSON data without a schema. This is a NoSQL database.
- **SQL database**: Used to build and update tables.
- **Power BI**: Used to display output directly in charts and tables.
- **Event Hubs**: Used to provide a high-volume telemetry buffer to co-located Azure services.
- **Service Bus**: Used to provide buffers to other processes. Service Bus queues and topics handle somewhat less volume than Event Hubs, but have more facilities for coupling to and from processes outside Azure or at other locations.

The third piece of a Stream Analytics job is the query code that transforms the inputs into the outputs.

Stream Analytics queries look superficially like conventional SQL queries, but there are some important differences. In particular:

- The `FROM` and `INTO` clauses of a `SELECT` statement can reference input and output connections that you have set up. A query can have more than one of each.
- A Stream Analytics job runs continuously until you stop it, and its queries are running continuously in parallel. Nevertheless, there's a **Test** function that allows you to run a query over a finite input table.
- No schema is required. Stream Analytics works with whatever it finds in the inputs. You can use path notation to access the fields of structured data, such as `Trip.StartPoint.Latitude`.
- The query language has some constructs that are concerned with timing. For example, there's a `LAG` expression that compares the current record with previous ones.
- There's no equivalent of aggregating over a whole database, but you can aggregate over arbitrary periods of time. For example, to average a value over successive periods of 10 seconds:

```
SELECT AVG(speed) as avg_speed GROUP BY TumblingWindow(second, 10)
```

To give a couple more examples, this query passes records straight from an input stream to an output:

```
SELECT *
INTO OutputStream1
FROM InputStream1
```

A more conventional structure separates the query into selecting the input fields and then computing the outputs:

```
WITH Data AS ( SELECT speed, rpm FROM CarInputData)
SELECT speed, rpm, speed/rpm as gear FROM DATA INTO PBIoutput
```

You don't have to test your queries on continuously streaming data. Instead, the Test feature (Figure 6-6 on the next page) lets you upload sample data and run on that. Currently, you have to use the Azure classic portal (https://manage.windowsazure.com/) to run the tester. It gives you access to all the same resources and data, but with a different user interface.

For some sample data, see HelloWorldaASA-InputStream.json. For a selection of example queries, see the "Stream Analytics Query Examples" in Chapter 10, *Reference*, along with query examples for common Stream Analytics usage patterns.

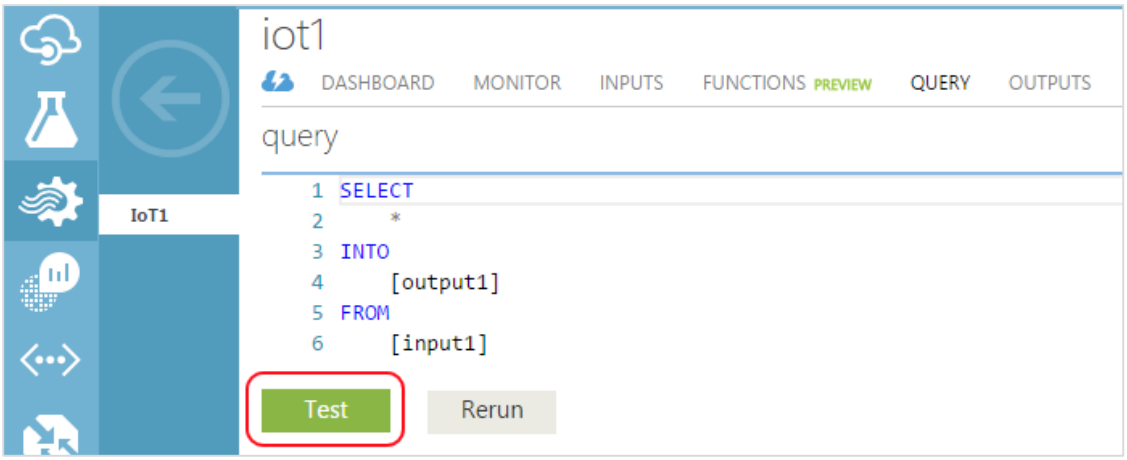


Figure 6-6: Testing Stream Analytics queries in the classic Azure portal

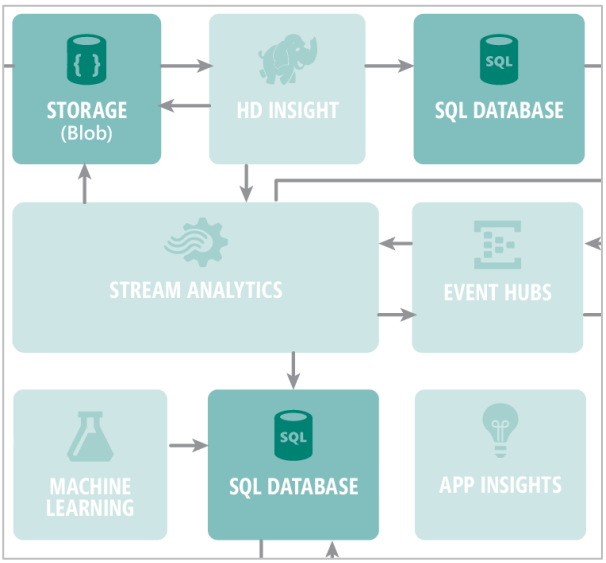
Best practices

To minimize long-distance traffic and latency, run Stream Analytics in the same geolocation as the other components. Output storage in particular should be co-located.

Where scaling is concerned, monitoring metrics are surfaced in the **Monitor** tab of the Azure Stream Analytics job. The main metrics show counts of input events, and error counts. As with other Azure metrics, you can set alerts so that you receive an email if any metric goes above a threshold you set.

As your traffic increases, you can scale up the power of your Stream Analytics job, in units of about 1 Mb/s of throughput per query step. You can also set up multiple event hubs and Stream Analytics units in parallel-running partitions.

Primer: Azure Storage



Within the MyDriving architecture data flows, we use Azure cloud storage as a medium- to long-term store for raw and analyzed data. Azure storage provides a variety of options and is massively scalable—you can use as little or as much as you want, and pay only for the data you store. Data storage options include:

- **Blobs**: For unstructured data such as documents or media. This is what MyDriving uses for Stream Analytics input and output.
- **Tables**: For NoSQL data (structured without a schema).
- **Queues**: For messages typically consumed in order.
- **Files**: For old applications that don't require any code changes.

It's easy to create a storage account in Azure. Select **New > Data + Storage > Storage account**, and then select a deployment model (Classic or Azure Resource Manager), as shown in Figure 6-7.

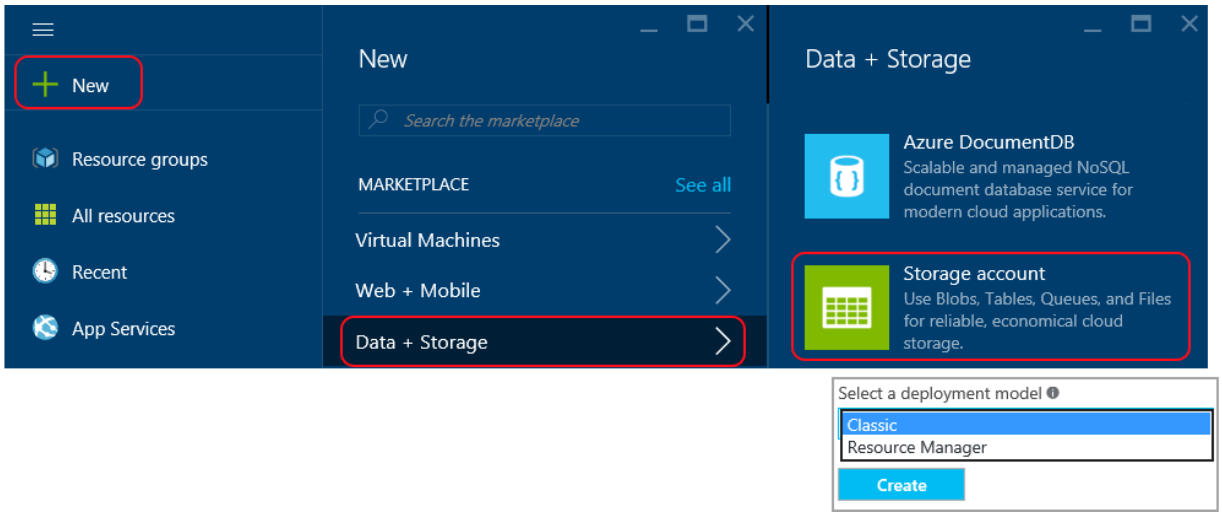


Figure 6-7: Creating new storage in the Azure portal

When you press **Create**, Azure prompts you for details such as the name for the storage, your subscription, resource group, and geolocation. After Azure deploys the storage, you can open it in the portal and configure its settings.

Within **Settings > Keys** (Figure 6-8, next page) you can give the account a name and see the access keys and connection strings that Azure generates automatically. These values are what you typically use in other Azure services to connect to that storage. In a Stream Analytics job, for example (Figure 6-9, next page) you can create a new input or output and paste the storage name and the access key.

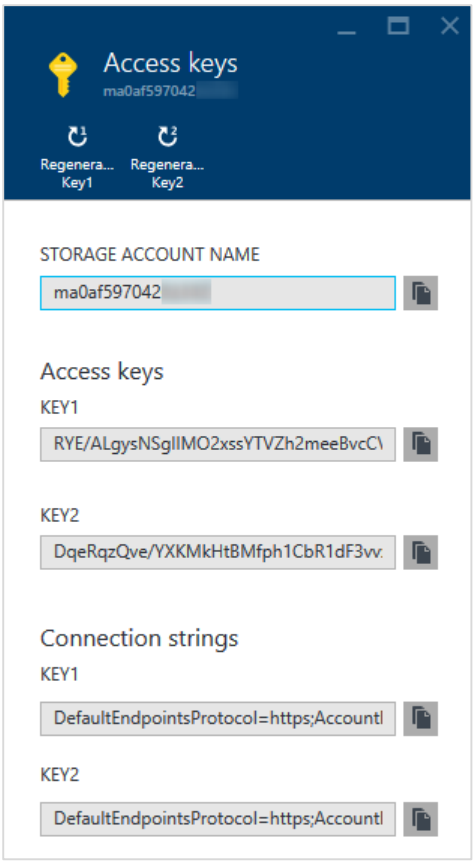


Figure 6-8: Access keys blade in the Azure portal, showing access keys and connection strings for a storage account

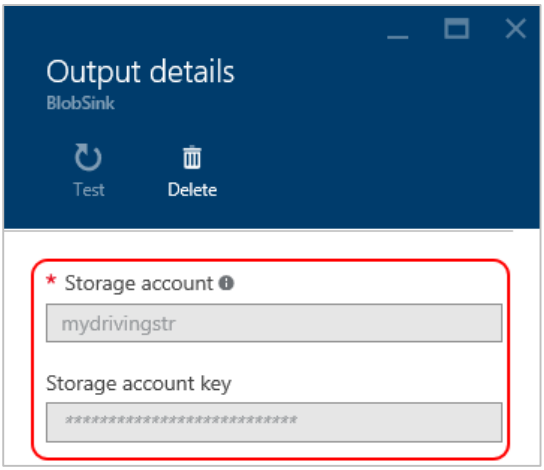


Figure 6-9: Setting the storage account and access key in a Stream Analytics job output; the fields here are disabled because the job is running

For other services you can do something similar, or use the connection string from code, such as an App Service API implementation or a microservice in Azure Service Fabric as we'll see in Chapter 9.

In general, Azure storage plays a central role in connecting multiple services together, as we'll see next.

Stream Analytics in MyDriving

MyDriving runs several different jobs in Stream Analytics as part of the different data flows. All of these connect to the IoT data stream coming from IoT Hub, but run different queries and send data to different sinks, as summarized in the following table.

Job	Sink name	Usage
`mydriving-sqlbi` (real-time hot path)	`PowerBISink`	Real-time aggregate data for consumption by Power BI.
	`SQLSink` (mydrivingDB SQL database)	Real-time aggregate data for consumption by App Service APIs (and thus the mobile app).
`mydriving-hourlypbi` (real-time hot path)	`PowerBISink`	A second hourly aggregation of real-time data for Power BI.
`mydriving-vinlookup` (extensions)	`EHSink`	Runs a simple query to extract the VIN number from a trip, and sends to Event Hubs for extensibility (see Chapter 9).
`mydriving-archive` (aggregate cold path)	`BlobSink`	Sends real-time data to blob storage where it accumulates for longer periods of time, for later processing by HDInsight and visualization in Power BI and for retraining Machine Learning. This is used for the historical or "cold path" flow, as described in Chapter 8, *Historical Data Handling*.

For `mydriving-sqlpbi`, which is the primary job for the real-time data flow, the job *topology* is shown in Figure 6-10 as it appears in the Azure portal with one input and two outputs. (The topology of `mydriving-hourlypbi` is similar, with just one output).

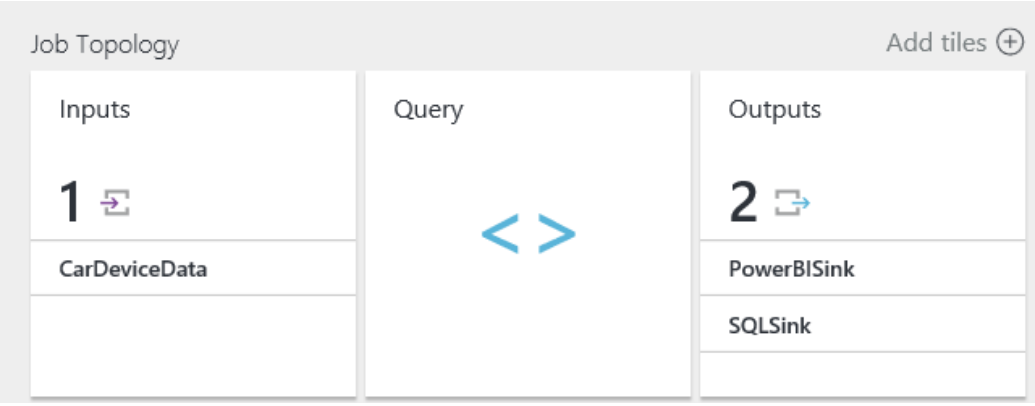


Figure 6-10: The mydriving-sqlpbi job topology

In any job, you can click the icons next to **1** or **2** to add additional inputs or outputs. In the `mydriving-sqlpbi` job, the `CarDeviceData` input is linked directly to the IoT hub via its shared access policy name and key as shown in Figure 6-11 on the next page.

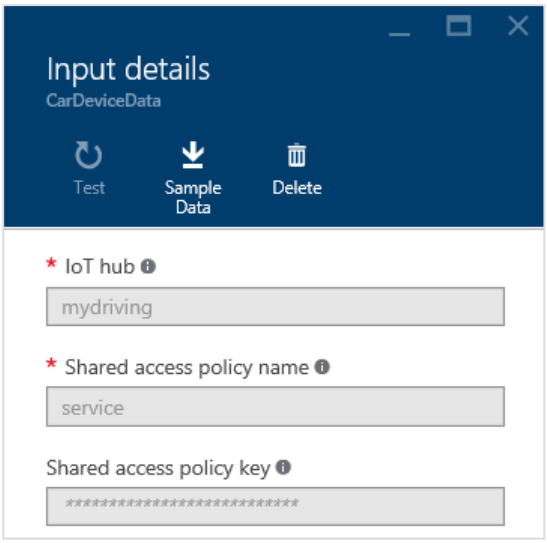


Figure 6-11: Details for the CarDeviceData input showing the connection to IoT Hub

Clicking on **Query** in the topology opens the query editor. For the `mydriving-sqlpi` job, you can find the query source code onGitHub, under src/StreamAnalytics in asa_sqlsink_powerbisink_query.txt.

The query begins with a simple shaping of the incoming IoT data:

```
WITH TripPointRaw as
(
SELECT
    TripId,
    UserId,
    TripDataPoint.Lat as RawLat,
    TripDataPoint.Lon as RawLong,
    CAST(TripDataPoint.Speed as FLOAT) as spd,
    CAST(TripDataPoint.EngineRPM as FLOAT) as enginerpm,
    CAST(TripDataPoint.EngineLoad as FLOAT) as engineLoad,
    CAST(TripDataPoint.ShortTermFuelBank1 as FLOAT) as shortTermFuelBank,
    CAST(TripDataPoint.LongTermFuelBank1 as FLOAT) as longTermFuelBank,
    CAST(TripDataPoint.MAFFlowRate as FLOAT) as flowRate,
    CAST(TripDataPoint.ThrottlePosition as FLOAT) as throttlePos,
    CAST(TripDataPoint.Runtime as FLOAT) as runtime,
    CAST(TripDataPoint.DistanceWithMIL as FLOAT) as distanceWithMIL,
    CAST(TripDataPoint.RelativeThrottlePosition as FLOAT) as relativeThrottlePos,
    CAST(TripDataPoint.OutsideTemperature as FLOAT) as outsideTemperature,
    CAST(TripDataPoint.EngineFuelRate as FLOAT) as engineFuelRate,
    TripDataPoint.RecordedTimeStamp as actualTS,
    DATEADD(millisecond,- DATEPART(millisecond,TripDataPoint.RecordedTimeStamp),
        DATEADD(second, 5 - CAST(CEILING(DATEPART(second, TripDataPoint.RecordedTimeStamp)%5)
        as BIGINT),TripDataPoint.RecordedTimeStamp)) as ts,
    DATEDIFF(millisecond, TripDataPoint.RecordedTimeStamp,
        DATEADD(millisecond,-DATEPART(millisecond,TripDataPoint.RecordedTimeStamp),
        DATEADD(second, 5 - CAST(CEILING(DATEPART(second, TripDataPoint.RecordedTimeStamp)%5)
        as BIGINT),TripDataPoint.RecordedTimeStamp))) as tsDiff,
    TripDataPoint.VIN as vin,
    TripDataPoint.RelativeThrottlePosition as throttle
FROM
    CarDeviceData TIMESTAMP by TripDataPoint.RecordedTimeStamp
WHERE
    TripId is not null
    and TripId != ''
    and UserId is not null
    and UserId != ''
),
```

This becomes the source of an aggregation, which maintains minimums, maximums, and averages:

```
TripPointAgg as
(
SELECT
    TripId,
    UserId,
    vin,
    ts,
    AVG(RawLat) as lat,
    AVG(RawLong) as lon,
    MIN(tsDiff) as lastRecTime,
    MAX(tsDiff) as firstRecTime,
    MIN(spd) as minSpeed,
    MAX(spd) as maxSpeed,
    AVG(spd) as avgSpeed,
    AVG(engineLoad) as avgEngineLoad,
    AVG(shortTermFuelBank) as avgShortTermFuelBank,
    AVG(longTermFuelBank) as avgLongTermFuelBank,
    MAX(enginerpm) as maxEngineRpm,
    AVG(flowRate) as avgFlowRate,
    AVG(throttlePos) as avgThrottlePos,
    MAX(runtime) as maxRuntime,
    MAX(distanceWithMIL) as maxDistanceWithMIL,
    AVG(relativeThrottlePos) as avgRelativeThrottlePos,
    AVG(outsideTemperature) as avgOutsideTemperature,
    AVG(engineFuelRate) as avgEngineFuelRate
FROM
    TripPointRaw
WHERE
    ts is not null
GROUP BY
    TripId,
    UserId,
    vin,
    ts,
    TumblingWindow(second,5)
),
```

Both are then used to create rough driving statistics:

```
RoughDrivingStats as
(
SELECT
    t1.TripId,
    t1.UserId,
    t1.lat,
    t1.lon,
    CASE
        WHEN t3.spd - t2.spd > 50 THEN 2
        WHEN t2.spd - t3.spd > 60 OR t1.maxSpeed - t1.minSpeed > 70
            OR t1.maxSpeed - t1.avgSpeed > t1.avgSpeed - t1.minSpeed
            + 0.10*(t1.maxSpeed - t1.minSpeed) THEN 1
        ELSE 0
    END as POIType,
    t1.ts,
    t1.avgSpeed,
    t1.minSpeed,
    t1.maxSpeed,
    t1.avgEngineLoad,
    t1.avgShortTermFuelBank,
    t1.avgLongTermFuelBank,
    t1.maxEngineRpm,
    t1.avgFlowRate,
    t1.avgThrottlePos,
    t1.maxRuntime,
    t1.maxDistanceWithMIL,
    t1.avgRelativeThrottlePos,
    t1.avgOutsideTemperature,
```

```
    t1.avgEngineFuelRate,
    t3.spd as firstSpeed,
    t2.spd as lastSpeed
FROM TripPointAgg t1
JOIN TripPointRaw t2
ON t1.TripId = t2.TripId and
t1.vin = t2.vin and
t1.ts = t2.ts and
t1.lastRecTime = t2.tsDiff and
DATEDIFF(minute,t1,t2) BETWEEN 0 and 0
JOIN TripPointRaw t3
ON t1.TripId = t3.TripId and
t1.vin = t3.vin and
t1.ts = t3.ts and
t1.firstRecTime = t3.tsDiff and
DATEDIFF(minute,t1,t3) BETWEEN 0 and 0
)
```

From this result, finally, we have the queries that route specific data into the two sinks:

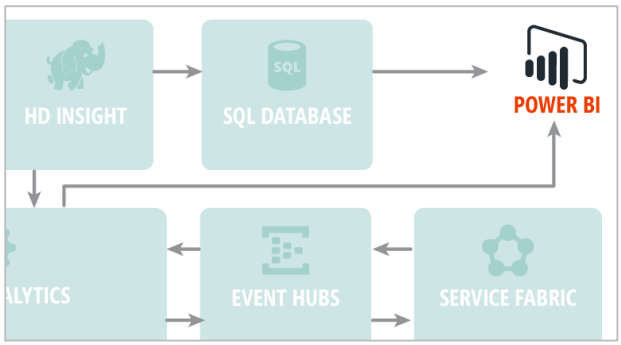
```
SELECT
    TripId,
    lat as Latitude,
    lon as Longitude,
    POIType,
    ts as RecordedTimeStamp
INTO SQLSink
FROM RoughDrivingStats
WHERE POIType > 0

SELECT
    TripId,
    UserId,
    avgEngineLoad as EngineLoad,
    avgShortTermFuelBank as ShortTermFuelBank1,
    avgLongTermFuelBank as LongTermFuelBank1,
    maxEngineRpm as EngineRPM,
    avgSpeed as Speed,
    avgFlowRate as MAFFlowRate,
    avgThrottlePos as ThrottlePosition,
    maxRuntime as Runtime,
    maxDistanceWithMIL as DistancewithMIL,
    avgRelativeThrottlePos as RelativeThrottlePosition,
    avgOutsideTemperature as OutsideTemperature,
    avgEngineFuelRate as EngineFuelRate,
    lat,
    lon,
    ts as RecordedTimeStamp,
    POIType
INTO PowerBISink
FROM RoughDrivingStats
```

This extensive query (as well as the asa_hourlybpi_query used for the `mydriving-hourlypbi` job) gives you a good idea of what Stream Analytics can really accomplish in a solution like MyDriving. Instead of writing a custom service in code to do this kind of data processing, we write only a query that instructs Stream Analytics to do the exact work required. All we need to concern ourselves with is the query, which is to say, the shaping of the data. Stream Analytics takes care of all other matters, like listening for incoming data, handling concurrency issues, and storing data in the appropriate outputs. This allows you to set up jobs like this in a very short time.

In fact, notice that we've said nothing more about routing the output data to the App Service API endpoints, from which it can be consumed by and visualized in the mobile app. This is because there's nothing more to be said! As soon as the Stream Analytics job runs for every piece of data that comes through IoT Hub, the results are *immediately* available through the APIs that go directly into the storage tables. This means that within a short time after you finish recording a trip in the MyDriving app—basically as long as it takes the job to run, which isn't very long at all—you can review that trip in the app and get results from Stream Analytics.

Nothing else needs to be said either about routing data to extensions in Event Hubs and Service Fabric, as we'll see in Chapter 9, or to Power BI. Again, as soon as the job is complete, the results are available in the applicable output sinks. In the case of Power BI, it immediately has new data to visualize, as we'll see in the next section.

Power BI



In the MyDriving solution, Power BI is used to visualize both real-time and historical data across the whole system. In this section we briefly introduce Power BI and its capabilities, then see how it's used in the hot data path. Power BI also comes up in Chapter 8 in regards to historical data.

Primer: Visualizing data with Power BI

Microsoft Power BI is a cloud-based, business intelligence and analytics service connecting users to a broad range of data through live streaming dashboards, highly interactive reports, and easy sharing and collaboration. Power BI includes the core service, Power BI desktop, and Power BI developer with visualizations on any device as represented in Figure 6-12.

Figure 6-12: Power BI on various devices

Features of Power BI core service:

- Ability to drill through to underlying reports.
- Natural language query with Q&A—ask questions of your data more naturally.
- Cortana integration—allows you to access your data from Windows 10.
- Quick insights—automatically discover patterns and insights in your data.
- Ability to easily connect to multiple data sources, bringing disparate data sources together.
- Native apps for iPad, iPhone, Android, and Windows devices.
- Embed fully interactive visuals in custom apps, web pages, and blogs.

Features of Power BI desktop:

- Free downloadable desktop application.
- Ability to connect to any data.
- Advanced data query, shaping, and modeling.
- Custom metrics with Data Analysis Expressions (DAX).
- Intuitive drag-and-drop report creation.
- Quick and easy publishing to the Power BI service.

Features of Power BI developer:

- Azure Stream Analytics for real-time data streaming.
- Power BI REST API for real-time data streaming.
- Custom visualizations support through the open source visualization framework.

Power BI in the real-time data flow

As noted earlier in this chapter, the `mydriving-sqlpi` Stream Analytics job outputs its aggregated IoT data to a Power BI output sink. When creating a Power BI sink in a Stream Analytics job, you're prompted for the Power BI account to use as shown in Figure 6-13.

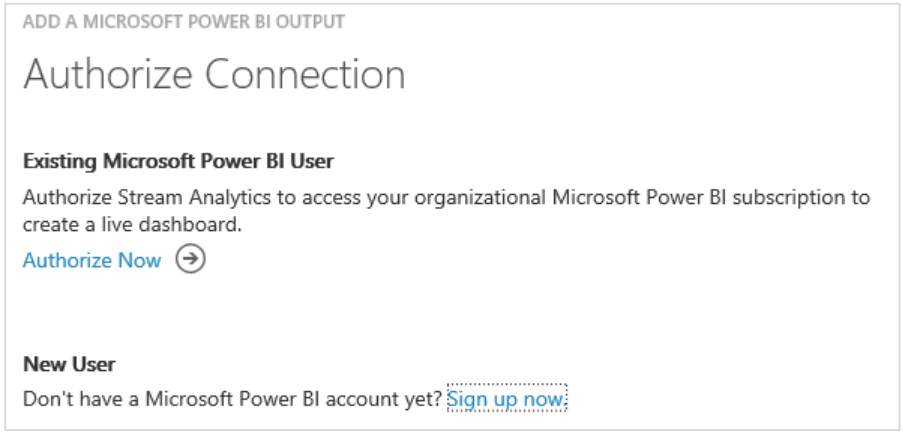


Figure 6-13: Connecting a Power BI account to Azure Stream Analytics

When the job is first run, the dataset is created for this account and appears in that user's workspace navigation pane in Power BI. The dataset is not explicitly available to other users in the organization; however, the account owner can share the dataset through a content pack, or create dashboards that can then be shared to other Power BI users.

In MyDriving, the dataset is again called `MyDriving-ASAdataset`. The dataset contains a schema, defined by the output query (table, fields), as well as the latest values for each data point in the output. Streaming data into the dataset is first-in, first-out; that is, each data point is retained in the dataset only until it is replaced by a new value. In MyDriving, the dataset contains only a single table, `TripPointData`, as defined in the query shown in Figure 6-14.

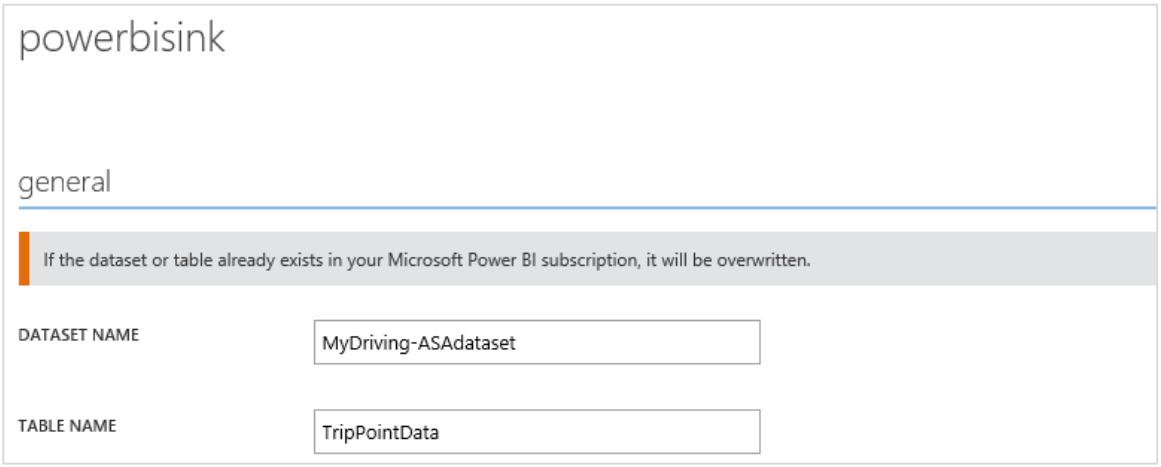


Figure 6-14: The configuration of a Power BI sink for Stream Analytics

It's from this dataset that you begin exploring the data in it by creating reports with visualizations in Power BI. One or more reports can be created for a dataset. Each report can contain one or more pages, and each page can contain one or more visualizations.

To create a report, select a dataset in the workspace navigation pane, which opens a blank editor canvas. In the editor (shown in Figure 6-15 on the next page for the MyDriving dashboard), the `TripPointData` table appears with all the fields as again defined by the query, including `TripId`, `UserId`, `EngineLoad`, `EngineRPM`, `Speed`, and so on. To create a visualization, click a field or drag it to the canvas.

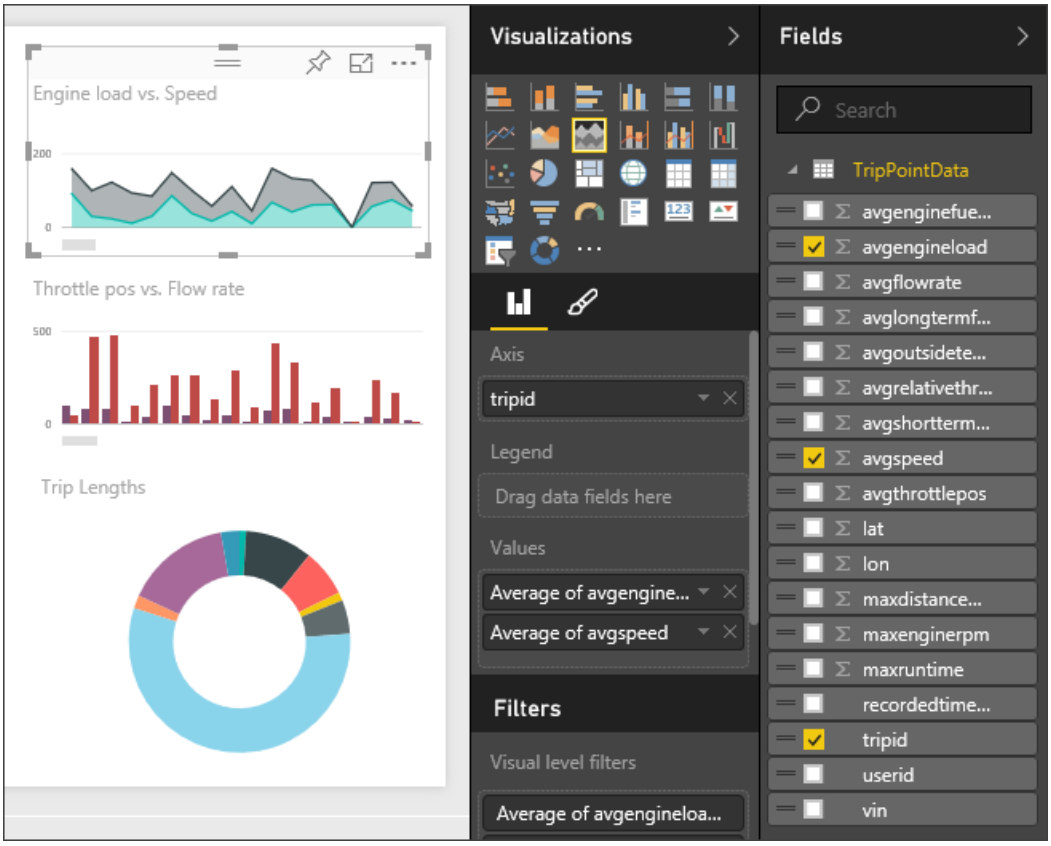


Figure 6-15: Creating a visualization in Power BI with the MyDriving TripPointData coming from Stream Analytics

Power BI has more than two dozen native visualizations. Not all visualization types work with any type of data. With streaming data, it's important to choose a visualization type that provides a quick, intuitive view of constantly changing values. For example, a chart visualization will not effectively display a single, changing value; however, a gauge chart will show that value in a format in which people can easily see the change, just as a speedometer shows the speed you are driving.

You can apply filters to an individual visualization or to all visualizations on a report page. Visualizations can then be pinned to one or more dashboards like that shown in Figure 6-16. Visualizations on dashboards are known as tiles. When a user clicks a pinned tile in a dashboard, the underlying report opens in the browser. Individual tiles or entire dashboards can also be shared with other Power BI users in the organization. You can also embed tiles in custom apps, web pages, or blogs.

A dashboard can combine views of data from multiple sources in a single pane, providing a consolidated view across the organization, regardless of where the data lives. Each metric, or insight, is displayed on the dashboard as a tile as also shown in Figure 6-16 on the next page.

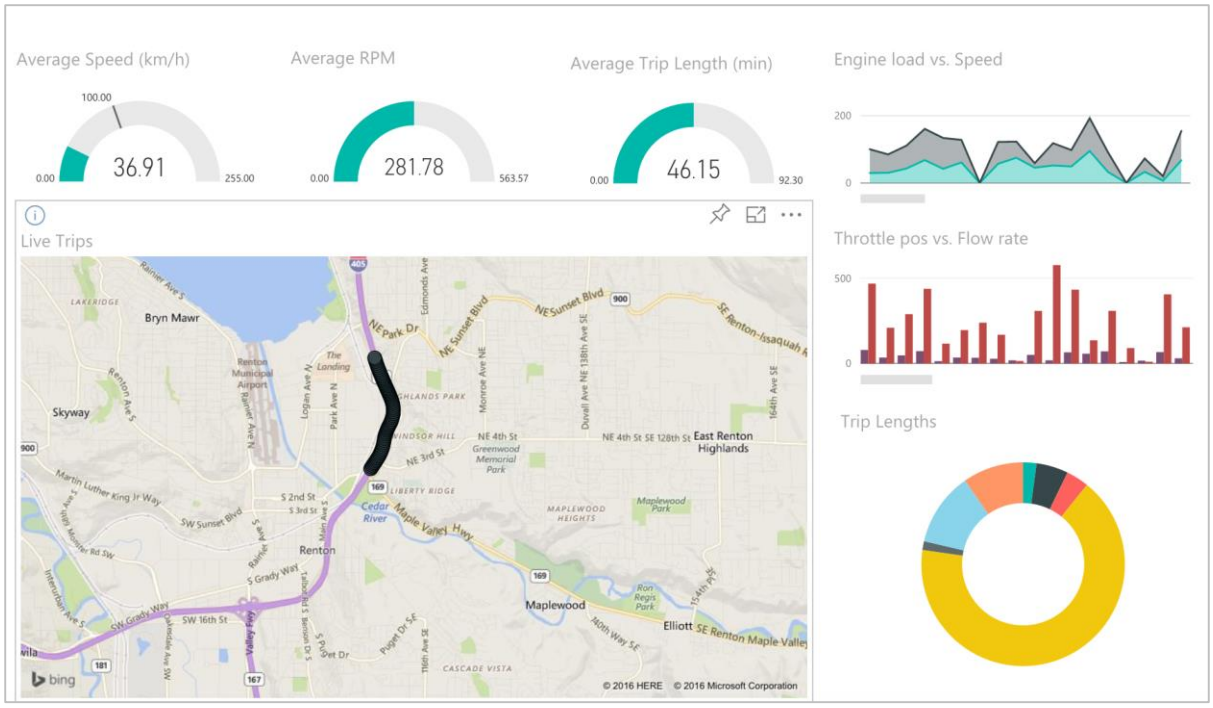


Figure 6-16: The MyDriving Power BI dashboard for real-time data

Additional notes

Power BI accounts. Power BI provides two types of accounts: Free and Power BI Pro. The account specified in a Stream Analytics output job should be a Power BI Pro account. A Power BI Free account limits the number of rows per hour that can be shown in stream visualizations. Data consumers, those users signed in to Power BI to view streaming data in reports and dashboards, also have constraints on the number of rows per hour that can be displayed.

Data consumers of historical, archived data, as we'll see in the Chapter 8, must be signed in to Power BI with a Power BI Pro account. This is because our historical data handling stores data in an Azure SQL database. Connections from the Power BI service to SQL database are live, and such connections are supported only for users with a Power BI Pro account.

> **Note** If the account you specify for the Stream Analytics job output has a password that expires, you will likely get an authentication error in the operations logs. To resume streaming, you must stop the job and renew authentication.

Throughput constraints. Power BI employs concurrency and throughput constraints, depending on user account type. You specify throughput (as a push event) in the query for the Stream Analytics output. For Power BI, use TumblingWindow or HoppingWindow to ensure data push is at most one push per second. In this scenario, our query defines data push as TumblingWindow, five seconds. That is, a data push to the dataset occurs once every five seconds.

Decision point. Power BI supports live streaming through Stream Analytics job output or through the Power BI REST API. For this architecture, a Stream Analytics output job is easily configured to authenticate a connection through a Power BI user account, create a dataset in Power BI, and output streaming data to the dataset specified in the query. The Power BI REST API is a good choice for custom apps, to push data directly from custom applications to a Power BI dataset.

Machine Learning

Machine Learning is used in MyDriving to highlight certain behaviors in one's driving habits. In terms of data flow, machine learning feeds its results to certain properties of the `Trip`, `UserProfile`, and `POI` (points of interest) tables in App Service, which are of course the data source for visualizations in the mobile app. The interesting part is that its training data comes from blob storage and HDInsight, rather than the real-time data stream, as shown in Figure 7-1, because HDInsight is the best service to aggregate historical data for this purpose.

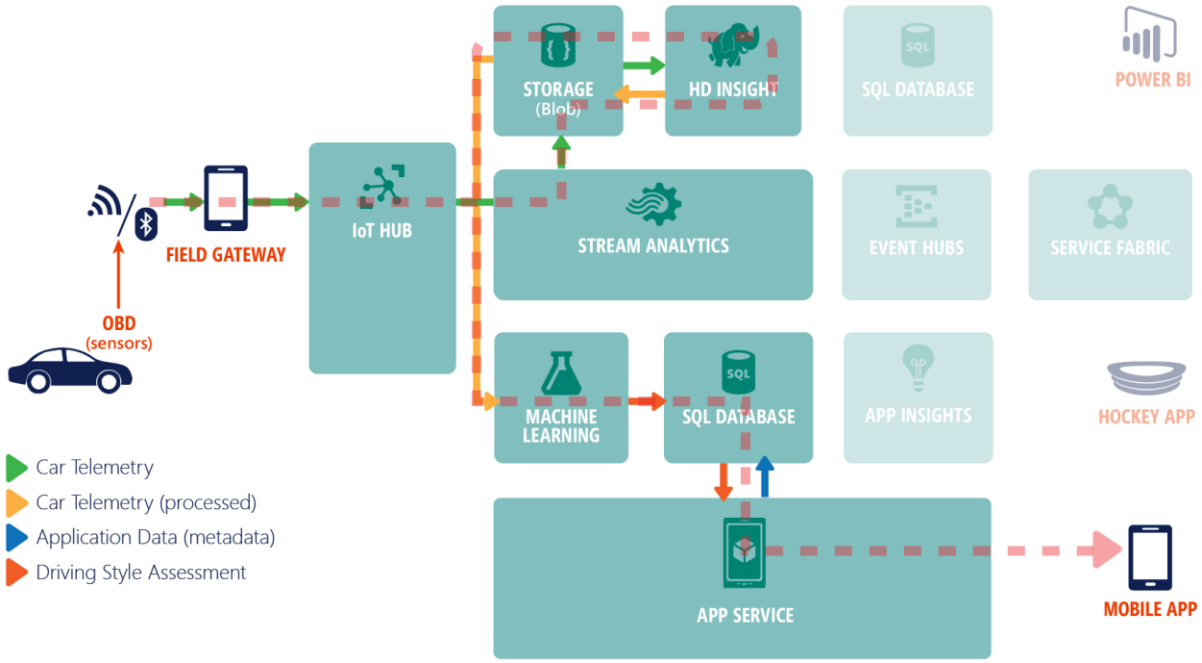


Figure 7-1: Machine learning in the MyDriving data flows

This chapter covers what Machine Learning is and what it does, as well as how it's specifically applied in MyDriving. For details about the other components in the flow diagram above, particularly HDInsight, see Chapter 8, *Historical Data Handling*.

Primer: Machine Learning

Machine Learning, which we'll refer to as ML for convenience, works in two modes: *training* and *prediction* (or *scoring*). In the training mode, a machine learning *model* is provided with data, and a *training algorithm* is applied to it.

For example, you might train a model with lots of data about the shapes and sizes and colors of a box full of fruit. Then in the *prediction* mode, the model is applied to new data. You might use it to look for anomalous fruits that don't have combinations of characteristics like the ones in the training box. Or, if you labeled the fruits in the original box, you could use the model to label new fruit based on shape and size.

Part of a Machine Learning workspace in Azure is shown in Figure 7-2. (Note that this isn't the one used in MyDriving—we'll come to that shortly.) It has tabs for two *experiments*—training and predictive. An experiment defines the data flow that you'll use to train and run the model. There's a toolbox of *modules* that you can pull onto the diagram and wire up. Along with a variety of training algorithms, there are modules for reading and preparing the data. You can also write your own modules using the R language.

Note For details about individual modules, see the A-Z List of Machine Learning Studio Modules.

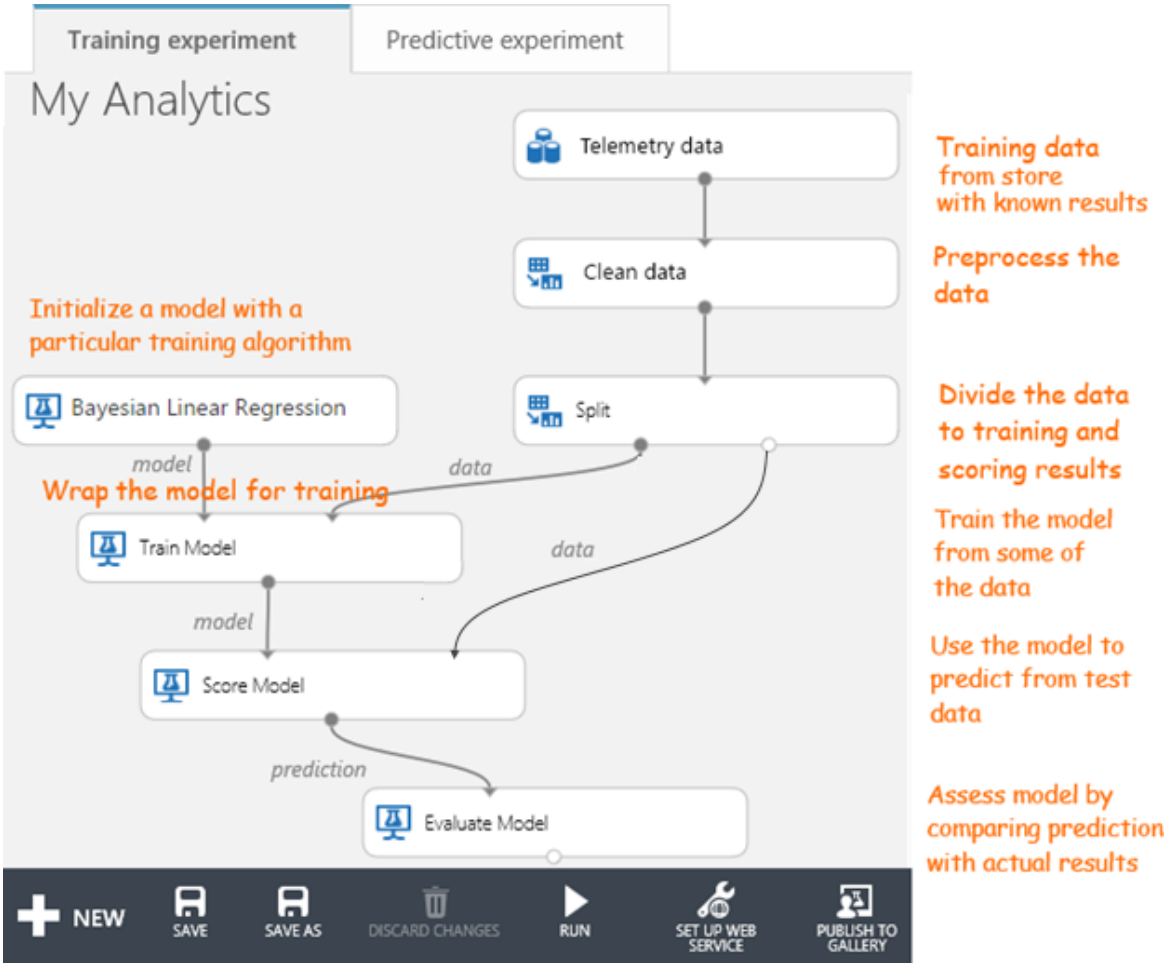


Figure 7-2: The Machine Learning workspace, with added labels in orange

The model isn't one of the boxes on the diagram, which represent processing modules. The model is a data type like any other, and is passed along some of the arrows from one module to another. In addition, it's passed through from the training experiment to the predictive experiment.

In Azure Machine Learning, the data that the model works on is a table of elementary values—strings, numbers, or enumerations. Each row represents a data point and is treated separately. If you want ML to deal with some relationship between successive points in your stream, you have to calculate the relationships before presenting them to ML. For example, you can calculate an acceleration column in your vehicle data by subtracting the speeds in successive data points.

Developing experiments

Don't be misled by the term "experiment": although developing an experiment is often a process of trial and error, some of your experiments will become a permanent part of your production system.

Training shouldn't be confused with the process of developing an experiment. Although you'll first train your model while you're in development, training can continue to happen frequently when your system is in production. Training and prediction are simply alternating modes of working with ML.

During development, you typically try out many different choices:

- **Choose the input data.** It might turn out that some input value (column) has little effect on outputs, like the size of a driver's hat on fuel cost. Because data points are distributed evenly over that input dimension, the model can't make any useful predictions based on it, and you can exclude that column. On the other hand, it might occur to you at this stage to collect data that's more likely to be useful, such as altitude and humidity for fuel costs.
- **Choose the training algorithm.** Different algorithms (training modules from the toolbox) are designed for different types of input and output. Further, different algorithms serve different objectives, such as anomaly detection or classification. Each algorithm has a number of parameters that can be adjusted as you experiment. (Learn more about available algorithms.)
- **Shape the input data.** Shaping is done with appropriate preprocessing, such as selecting different columns, computing new columns, or by aggregation. The toolbox includes a variety of preprocessing modules, and you can write your own using the R language.

During development, you usually use a fixed batch of sample data as your input. Running the model for training or prediction only takes a few seconds. You can look at the output of any module by right-clicking to see a table or graph.

Types of machine learning

Classification is a form of *supervised learning*. You might provide, for instance, a table of the sizes, colors, and other properties of a box of fruit, together with a *label* column. ML discovers the patterns in the data that correspond to the labels. Then you can get another box and ask ML to label the new fruit by inference from their properties, as illustrated in Figure 7-3 on the next page. It's "supervised" because during training, you're telling it the answers you want—that is, the labels.

Figure 7-3: Training provides answers for Machine Learning so that it learns to predict answers from partial data

Clustering is a form of *unsupervised learning*. Suppose you dump a pile of fruit into ML—well, data about the fruits' properties—and ask it to sort them into similar piles. It will find clusters such as small yellow, large yellow, small red, and so on, and label them with numbers (Figure 7-4).

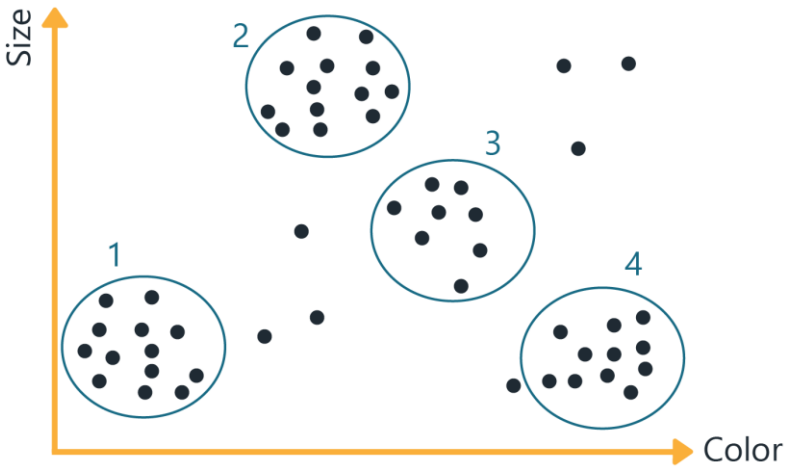


Figure 7-4: Clustering of input data

Of course, it's easy for human beings to look at such a two-dimensional diagram and see the clusters. But typically there are many more dimensions involved, and it's much harder for us to envision the data. That's a job that's done much better by computers!

There are various things you could do with a clustering result. One would be to identify anomalies in new fruit: for example, raise an alarm if we see a large blue fruit, because it doesn't fit a known cluster.

Using a combination of algorithms

Now suppose you have a lot of different types of fruit in your market, and you want to train Machine Learning to recognize all of them. Naturally, you have to provide enough columns and values in the input data to allow the different types to be distinguished, and you have to provide a lot of training samples to allow ML to discover how to classify them. Ideally, thousands of samples.

The drawback of classification is that it requires you to label each data point for training. Doing this with thousands of points would be tedious. So instead, you can use a two-stage plan:

1. Use clustering to identify the different clusters of properties. These will correspond to different kinds of fruit. ML will label each data point with the ID of its cluster. You can then examine the clusters and replace the cluster IDs with human labels, like "apple."
2. Train a classification model on the labeled fruits. That gives you a trained classification model that you can use to classify new fruit. It's this model that you use in prediction mode.

Now you're in a position to look at the ML workspace used in MyDriving.

Machine Learning in MyDriving

In MyDriving, our aim is to show some practical details of using Azure Machine Learning as part of a larger IoT solution. The role of Machine Learning in this example is to learn to identify different styles of driving, and then to assign one of those styles to each trip. To keep things simple, you'll we constrain the model to identify only two classes of style in the data, and use only a few columns.

The learning strategy uses the two-stage process we introduced in the previous section. As with the unlabeled fruit example, we start off with a large table of data about road trips, and use a clustering algorithm to identify different driving styles. Then we rename the cluster IDs with readable names, and train a classification model to recognize the style of any particular trip.

Figure 7-5 shows the first half of this process. At the top you can see the *Reader* module is what brings in the data. The data source here, however, is not the real-time data flow as you might expect, but rather the blob storage where the `mydriving-archive` Stream Analytics job saves such data for longer-term processing. Note that the MyDriving system is using HDInsight to process that archive data to produce appropriate training input for Machine Learning. (Chapter 8 covers this in more detail.)

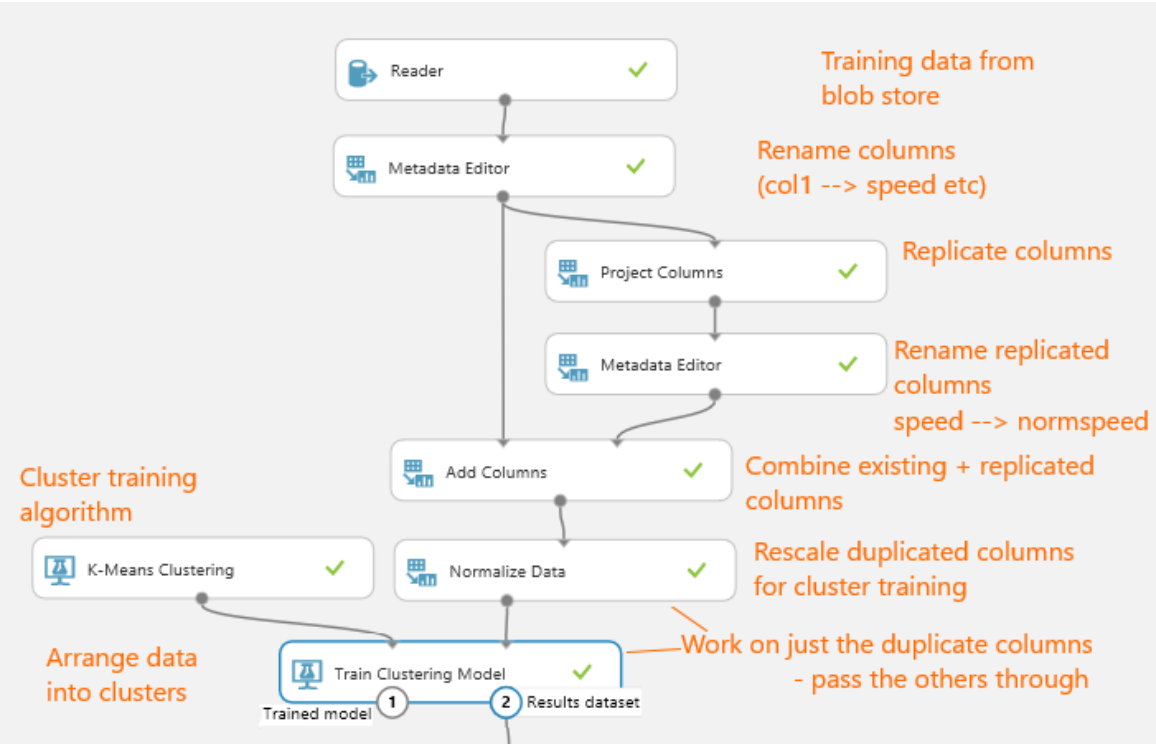


Figure 7-5: A machine learning process for clustering data. For details on the individual modules, see A-Z List of Machine Learning Studio Modules.

Train Clustering Model (the box at the bottom of the diagram) creates a model designed for clustering; it takes a plug-in algorithm, in this case the K-Means algorithm. This algorithm creates a number of *centroid* points, and then iteratively moves them around so as to minimize the sum of the distances between all the data points and their nearest centroids. In this project, the number of centroids is fixed at two.

Now we come to some practical considerations. A restriction of this algorithm is that it works best if the values are scaled and shifted so that their mean values in each dimension are 0. This is the purpose of the Normalize Data module.

However, this introduces a complication. What we want to look at eventually is the original data—not the normalized figures. We solve this by adding a copy of each of the columns that we want the clustering to work on, including acceleration and engine speed variation. We can tell the normalization and the clustering modules just to work on those duplicate columns and pass the others through. This gives us data points assigned to clusters according to the important dimensions, yet still allows the original values to persist.

After running the training experiment, we can see that the model has found two clusters. Right-click the output of the training module to get a visualization (Figure 7-6) of the principal components of the clustered data.

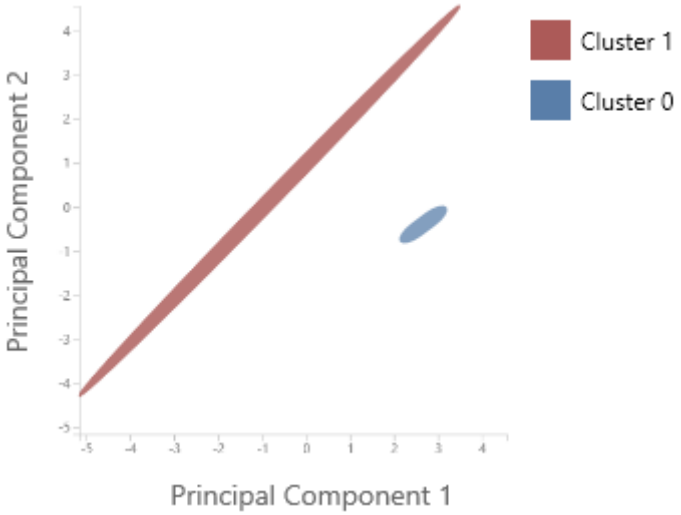
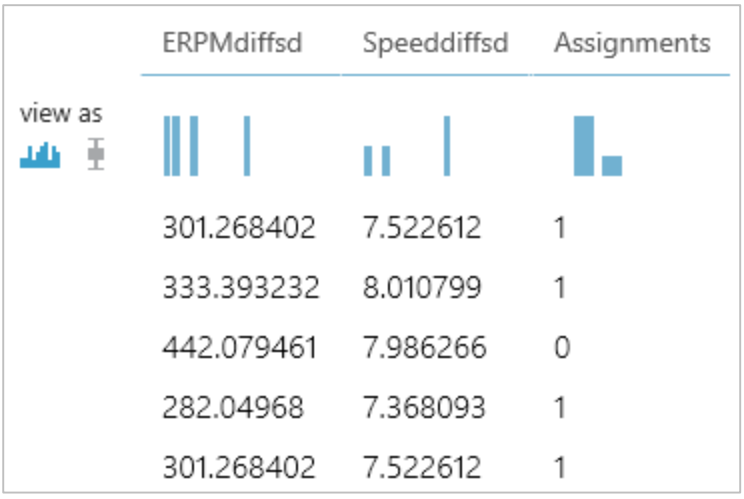


Figure 7-6: A visualization of training module output

Notice that the training module has two outputs: one for the model it trained, and the other for the clustered data. We won't actually use this model. The output data points are just as they went in, but with one extra column: the assignment of each point to a cluster. We can use a *Project* module to throw away the normalized columns, and just keep the important original columns, now with their assignments as shown in Figure 7-7 (next page).



	ERPMdiffsd	Speeddiffsd	Assignments
view as			
	301.268402	7.522612	1
	333.393232	8.010799	1
	442.079461	7.986266	0
	282.04968	7.368093	1
	301.268402	7.522612	1

Figure 7-7: Assignments for important original columns in the machine learning output

Naming the clusters: R-script

The clustering algorithm just assigns "1" and "0" as the labels for the driving-style clusters that it identified. Let's choose a pair of more readable labels, and pass the data through a module that will do the renaming. We can write a custom module for this, coded in the R language.

The *R Script* module allows us to code up our own data transformations. R is a widely used language in analytics. You can edit and test a script in an IDE such as R Studio before copying it into the module. (For more on R, see Quickstart tutorial for the R programming language for Azure Machine Learning.)

Click the R-Script module to see the code it encapsulates. Here are the essential bits:

```
count0 <- sum(dataset1$Assignments == 0)
count1 <- sum(dataset1$Assignments == 1)
...
if (count0 > count1) {
    dataset1$Assignments[dataset1$Assignments == "0"] <- "Good"
    dataset1$Assignments[dataset1$Assignments == "1"] <- "Bad"
} else {
   dataset1$Assignments[dataset1$Assignments == "1"] <- "Good"
dataset1$Assignments[dataset1$Assignments == "0"] <- "Bad"
}
```

As you can see, the script finds out which cluster is smaller and labels it as "bad." (You can decide on your own labels, of course.)

Training a classifying model

Now that we've identified and named some clusters in the historical data, we want to label new data in the same way. For this, we'll train a classification model to recognize the properties of data that have the different label values. This is the second part of our process, which is shown in Figure 7-8 on the next page.

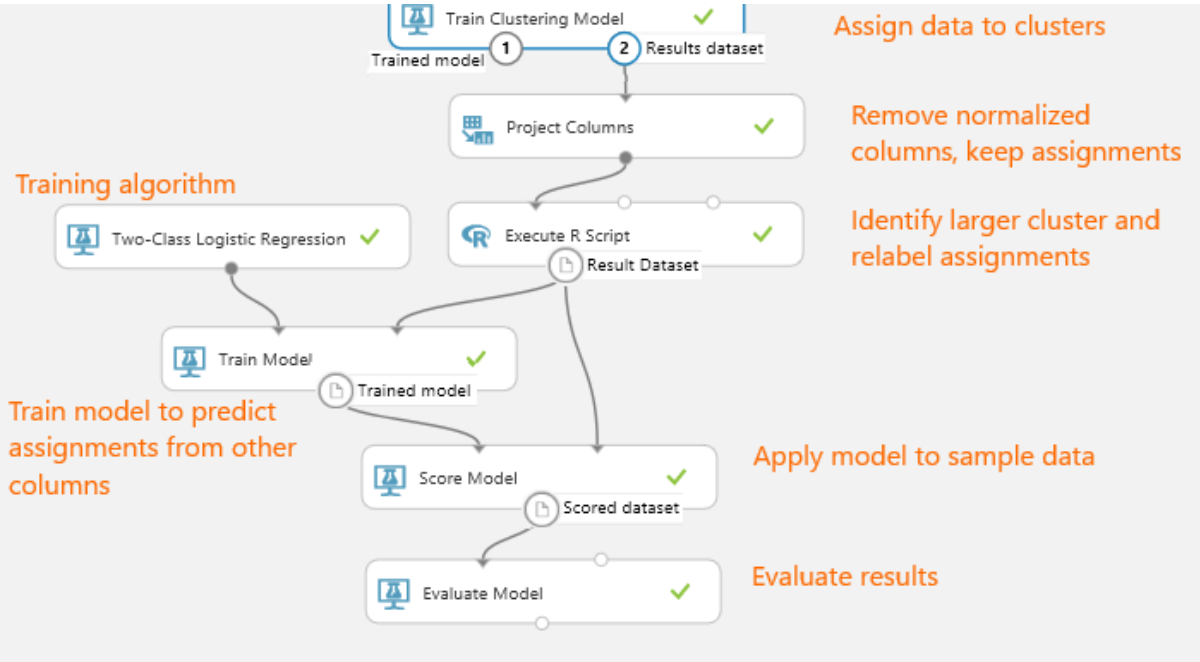


Figure 7-8: The second part of the machine learning experiment to train the classification model

Train Model specializes in supervised training, and has a parameter to specify the column we're using as a label. In this case, we specify the driving style column, created in the clustering step. *Score Model* applies the resulting model to the data.

This trained model is then saved from this experiment to run in the related Prediction experiment. Machine Learning preserves a link between the two experiments, so that the model can be retrained and automatically updated for prediction.

Predictive mode

In predictive mode, we don't need the clustering algorithm. Instead, we're just using the classification model as shown in Figure 7-9 (next page). If we present it with data for a road trip, it will answer with a driving-style label. (The trip data with which we query the model is in fact the same data that we store to use for future re-training sessions.) This is how Machine Learning effectively plugs into the real-time data flow to produce the `Trip.Rating` and `UserProfile.Rating` properties that are returned through the App Service API.

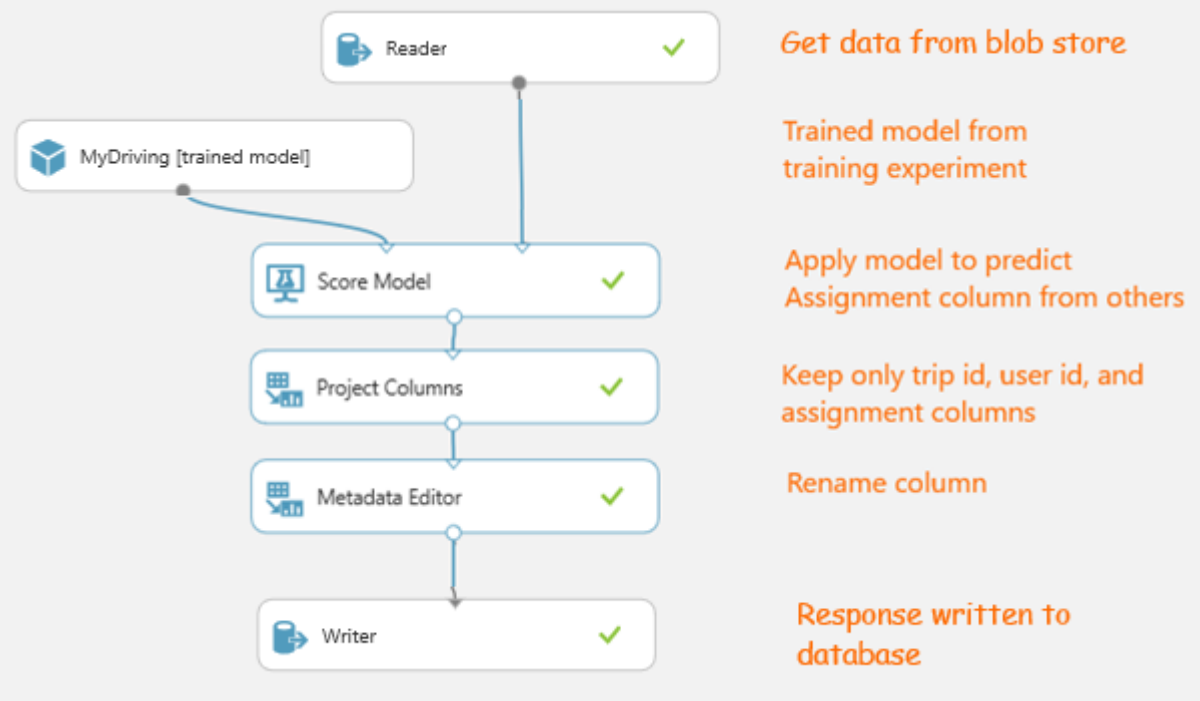


Figure 7-9: The predictive mode experiment for MyDriving

In this experiment, you can see that most of the modules have gone, but the two-class classifying modules from the training experiment remain. The scoring module, which is used to assign the driving-style label, is also present. The model classifies the data by recognizing patterns of properties that correspond to the labels on which it was trained. Although those labels were originally derived by identifying clusters, we're no longer performing any clustering in predictive mode.

In this system, the predictive system is run on a batch basis. Other parts of our system put partial data in blob storage, where Machine Learning picks it up periodically, and writes the results to a database. As you can see in the diagram, each query passes through the scoring module and is, with some adjustment to the columns, returned as the result of the query.

As an alternative, Machine Learning can provide a web interface, in which data is passed to the model in real time.

Retraining

The more input data, the more stable the model. As well as assessing each trip in real time, we store all the trips and use the accumulated trips to re-train the model. Retraining isn't a cumulative process; we simply create a new model from the training experiment each day, using all the data we've accumulated to date.

Notice also that we don't keep the labels of data points. In every re-training session, we go through the whole process of finding clusters, attaching labels, and training a new classification model.

To manage the retraining schedule, we use an Azure Data Factory. A data factory is a way of orchestrating the movement of data through multiple activities in a pipeline. The activities can be processes running within a service—which doesn't have to be in Azure. The data doesn't have to go through the data factory itself: its job is to trigger activities, either to a schedule or when data from one service becomes available to be picked up by another. So although the factory runs in an Azure process, it

can be used very effectively to move data between different services on your on-premises servers, as well as between Azure services.

In general, a data factory defines a *pipeline,* which connects a number of *activities,* which run in *services*. A service can be either in or outside Azure, and the pipelines can have multiple steps. You can edit the pipeline in a diagram, and it is defined by JSON scripts.

However, in this case, we're using the data factory to trigger just one process: the daily retraining of the Machine Learning model to refine it by using data collected during the day. Again, the retraining data comes from the archival blob storage that's been processed with HDInsight. That is the subject of the next chapter.

Historical Data Handling

Whereas the "hot data" flow we saw in Chapter 6, *Real-Time Data Handling* works with the data as it's streaming in from IoT Hub, the "cold path" or historical flow in the MyDriving system, shown in Figure 8-1, does most of its data processing and visualization at a later time.

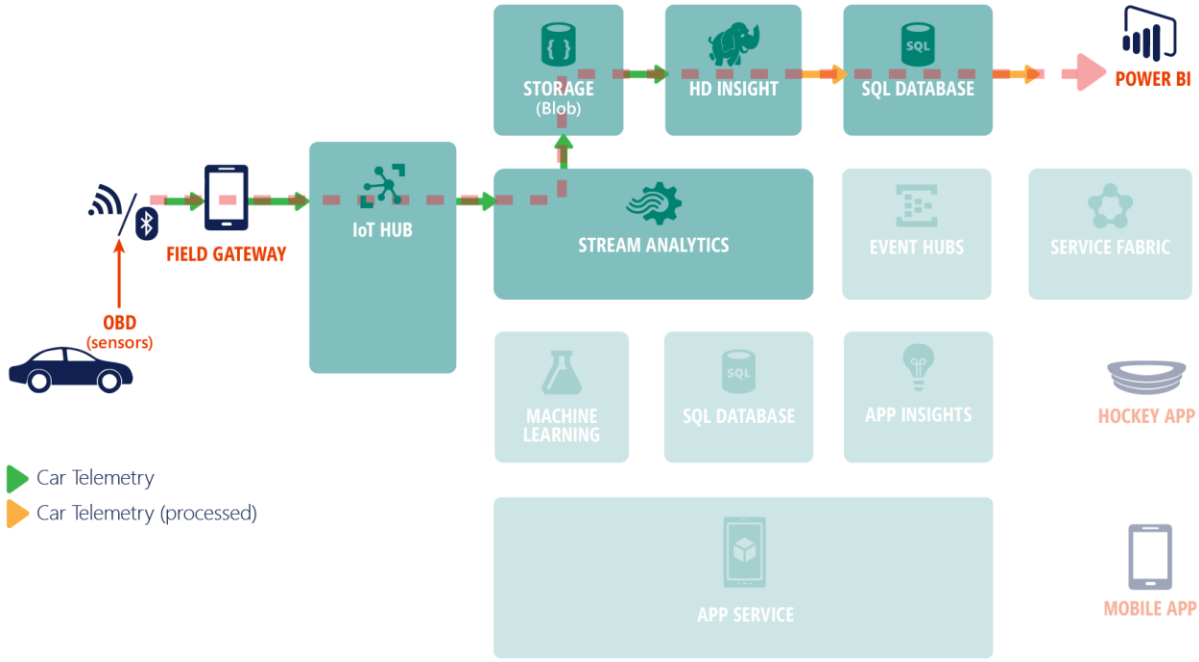


Figure 8-1: Historical data handling in MyDriving

This flow starts with the same real-time data that comes through IoT Hub that is again picked up by a Stream Analytics job, in this case the one named `mydriving-archive` (see "Stream Analytics in MyDriving" in Chapter 6). As with all other jobs, this one applies a query to the IoT data and sends it to storage, which in this case is an Azure blob storage account named `mydrivingstr`. The simple job topology and the linked output storage account is shown in Figure 8-2 (next page).

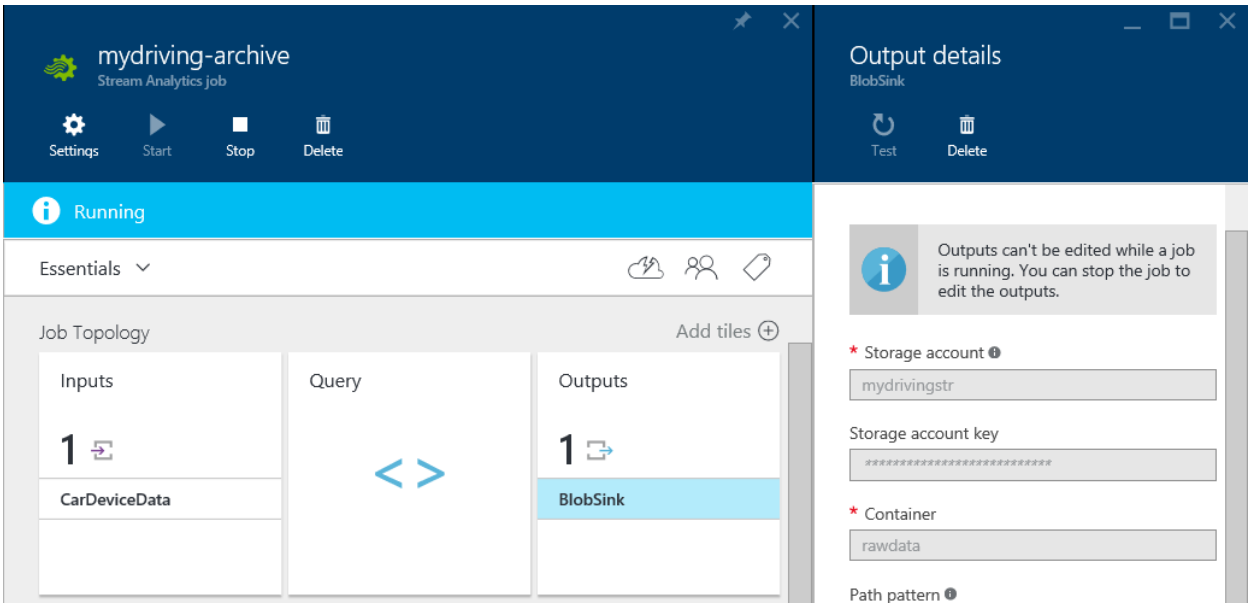


Figure 8-2: The topology of the mydriving-archive job and its output storage account

The blob storage thus accumulates large amounts of IoT data across all the connected devices. We can let this data accumulate really as long as we want at a fairly low cost, as is inherent with blob storage. But of course we want to make it available reasonably soon to visualize in Power BI.

To accomplish this, we periodically (about once a day) apply Azure HDInsight to shape the data into something more meaningful and consumable. The results from HDInsight go into an Azure SQL database, which serves as the data source for Power BI. The results also go back into blob storage for retraining the Machine Learning component that we saw in Chapter 7, *Machine Learning*. Thus in addition to the real-time monitoring of data in Power BI that we saw in Chapter 6, we now also have visualizations of the historical data (which might lag the real-time data by as much as a day, but not more).

Clearly, you can change the freshness of those historical visualizations in your own systems by changing the frequency of processing through HDInsight. How you present that data is entirely up to you and the needs of your scenario.

Because blob storage is merely employed here as a holding area for the potential flood of IoT data between processing sessions, we won't go into any more detail on that service here. You can learn the basics of blob storage, see the Introduction to Storage page in the Azure documentation.

What remains, then, is to look at how HDInsight works with the data, and how Power BI picks it up for visualization.

Primer: HDInsight

When you start working with data generated by an IoT solution, you are almost always working with "big data." Just think about the amount of data that would be generated by having a sensor in ten thousand cars reporting the vehicle status—speed, gas mileage, brake performance, etc.—every second. You need specialized analytics tools to turn this deluge of data into something meaningful and actionable. You need real-time analytics that can alert you to potential problems before they happen, and historical analytics that allow you to look for patterns and trends that emerge over time.

Handling big data also requires specialized tools for storing and analyzing large volumes of data, which can arrive in high volumes and may have variable structure or no structure at all. Although the final output of big data analysis may be stored in a traditional relational database, the analysis and storage of the raw incoming data must be handled using distributed, parallel processing in order to receive timely results.

One of the most well-known solutions for this processing is Apache Hadoop. Originally Hadoop provided distributed storage for big data, and used the MapReduce programming model to perform distributed parallel processing of historical data. MapReduce processes data by filtering and sorting data (the *Map* phase) and then summarizing the data (the *Reduce* phase). Higher level languages have then been implemented on top of MapReduce to reduce the complexity of implementing analytics as map and reduce functions. Two of the more popular solutions are Pig, which provides an easier way of implementing MapReduce logic through the Pig Latin language, and Hive, which provides a SQL-like language (HiveQL) for working with structured data.

Over time, Hadoop has evolved to become an open, pluggable architecture that allows other data analytics models beyond MapReduce:

- **Apache Spark**: Distributed, in-memory batch analysis that can be several orders of magnitude faster than traditional MapReduce solutions.
- **Apache Storm**: Distributed, real-time ingestion and analysis of streaming data.
- **Apache HBase**: Distributed, non-relational (column-oriented, key-value) database.

One of the challenges with Hadoop is creating and maintaining the "cluster," the computers or virtual machines that Hadoop uses for distributed storage and processing. Azure HDInsight is Microsoft's software as a service (SaaS) offering that provides Apache Hadoop, and software in the Hadoop ecosystem, as a managed service.

HDInsight reduces the complexity of working with big data analytics by providing clusters that are pre-configured for specific workloads, and that can be scaled dynamically to meet your workload requirements. For interoperability with other Azure services, and to allow the deletion of clusters when they are no longer needed, HDInsight stores data in either Azure Storage blobs or Azure Data Lake Store (an HDFS- and WebHDFS-compatible storage system that can hold data of any size and grows as you need more storage). For more information see the Data Lake Store product page.

HDInsight currently provides clusters that are tuned for the following workloads:

Workload	HDInsight cluster type
Batch processing (MapReduce)	Hadoop
Batch processing (in memory)	Spark
Real-time stream analytics	Storm
NoSQL data store	HBase

Each cluster type provides utilities and services required to the workload the cluster is tuned for. Core Hadoop technologies for moving, transforming, and cleaning data, such as Pig and Hive, are available on all cluster types.

HDInsight in MyDriving

In the MyDriving scenario, a Hadoop on HDInsight cluster (named `mydriving-hdi`) is used in the "cold data flow" for two purposes:

1. Transform the raw, unstructured data stored in the `mydrivingstr` blob storage account into structured data. This structured data can then be exported to SQL Server, in a storage account named `mydrivinglogs`, from which it is brought into Power BI for visualization.
2. Process raw data to produce suitable inputs for retraining the machine learning component, which are stored back into `mydrivingstr` blog storage.

To connect the input and output storage accounts, in the HDInsight cluster, select **Settings > Azure Storage Keys**.

> **Note** HDInsight and Storm could also be used for IoT data ingestion in place of Azure Stream Analytics, but the latter provides a simpler means to fulfill the requirements of the MyDriving scenario. Historical analysis, on the other hand, is not a streaming scenario, so HDInsight is better suited to process the large amounts of data in the blob storage.

The HDInsight cluster is created on-demand (to reduce costs) by an Azure Data Factory data pipeline, which also automates the Hive and Pig jobs run by HDInsight to process the data. The Data Factory Copy activity then moves the data to the SQL database. Once processing is complete, Data Factory deletes the HDInsight cluster, as it is no longer needed.

You can configure HDInsight in greater detail through the cluster dashboard, accessed through the HDInsight blade in Azure. The most important details here are the HiveQL and Pig scripts that really define HDInsight's behavior and, like Stream Analytics jobs, are quite similar to SQL. (See Hive with Hadoop and Pig with Hadoop in the HDInsight documentation.)

In MyDriving, four HiveQL scripts are running in the cluster. You can find these scripts in the src/HDInsight folder in the GitHub repository. Of these, CreateRawTable.hql is what first pulls in data from blog storage into a table called `tripdata`. This becomes an input for the factTripData.hql script that does the heavy lifting of shaping the output, including the identification of hard stops and hard accelerations, which is what feeds into the machine learning component. Here's that script in full:

```
DROP VIEW IF EXISTS tripDataWIPView;

CREATE VIEW tripDataWIPView as
SELECT TripId,
UserId,
vin,
MIN(unixtimestamp) as minUnixTimestamp,
AVG(cLat) as cLat,
AVG(cLon) as cLon,
AVG(AverageSpeed) as AverageSpeed,
MAX(Runtime) as TripRuntime,
MAX(DistanceWithMIL) as DistanceWithMIL
FROM tripDataInt
GROUP BY TripId, UserId, vin;


DROP TABLE IF EXISTS tripDataWIP;

CREATE TABLE tripDataWIP STORED AS ORC as
SELECT TripId,
UserId,
vin,
FROM_UNIXTIME(minUnixTimestamp) as tripStartTime,
cLat,
cLon,
```

```
AverageSpeed,
CAST(TripRuntime/60 as INT) as TripRunTime,
CASE
 WHEN DistanceWithMIL>0 THEN true
 ELSE false
END as DroveWithMIL
FROM tripDataWIPView;


-- Hard Brakes & Hard Accels
-- Assuming we're getting exactly 1 event per second per tripid, we are defining Hard
Accelerations and hard brakes over a 5-second period. Ideally, you'd have a more complex
definition and would likely need to create UDF to solve the problem.

DROP VIEW IF EXISTS speedCalc;

CREATE VIEW speedCalc as
SELECT
a.TripId,
a.unixtimestamp,
LAG(a.unixtimestamp,1,0) OVER (Partition BY a.TripId
    Order by a.unixtimestamp ASC) as prevTimestamp,
a.AverageSpeed as lastSpeed,
LAG(a.AverageSpeed,1,0) OVER (Partition BY a.TripId
    ORDER BY a.unixtimestamp ASC) as firstSpeed
FROM TripdataInt a;


DROP VIEW IF EXISTS harddrivingInt;

CREATE VIEW harddrivingInt as
SELECT TripId,
unixtimestamp,
IF ((lastSpeed - firstSpeed)*0.278/(unixtimestamp - prevTimestamp)>=6,1,0) as hard_accel,
IF ((lastSpeed - firstSpeed)*0.278/(unixtimestamp - prevTimestamp)<=-6,1,0) as hard_brake
FROM speedCalc;


DROP TABLE IF EXISTS harddriving;

CREATE TABLE harddriving STORED as ORC as
SELECT a.TripId,
SUM(a.hard_brake) as Hard_Brake,
SUM(a.hard_accel) as Hard_Accel
FROM harddrivingint a
GROUP BY a.TripId;


DROP TABLE IF EXISTS tripdataFinal;

CREATE TABLE tripdatafinal
(
TripId string,
UserId string,
vin string,
tripStartTime string,
AverageSpeed double,
Hard_Accel int,
Hard_Brakes int,
DroveWithMIL boolean,
LengthOfTrip int,
cLat double,
cLon double
) ROW FORMAT DELIMITED FIELDS TERMINATED BY '|' LINES TERMINATED BY '\n'
STORED AS TEXTFILE
LOCATION 'wasb://tripdata@mydrivingstr.blob.core.windows.net/tables/factTripDataoutput';


INSERT INTO TABLE tripdatafinal
SELECT a.TripId,
a.UserId,
IF(a.vin is NULL OR a.vin=='', "Unknown", a.vin) as vin,
```

```
a.tripStartTime,
a.AverageSpeed,
b.Hard_Accel,
b.Hard_Brake,
a.DroveWithMIL,
a.TripRuntime,
a.cLat,
a.cLon
FROM tripDataWIP a JOIN harddriving b
ON a.TripId = b.TripId;
```

As with Stream Analytics, you can see that one query fed into HDInsight can perform detailed data shaping and analysis that easily scales to big data proportions.

Power BI for historical visualizations

The final part of the historical data flow are the visualizations in Power BI. As described in Chapter 6, once you connect Power BI to a data source, you simply use its visualization designers to create the reports you're most interested in. For example, here's the main report we've created for MyDriving (Figure 8-3). As you can see, it displays much broader aggregate data across all the drivers and trips in the system. Of particular interest is the *Stop & gos by the hour* chart in the lower left, which clearly shows the impact of traffic on driving!

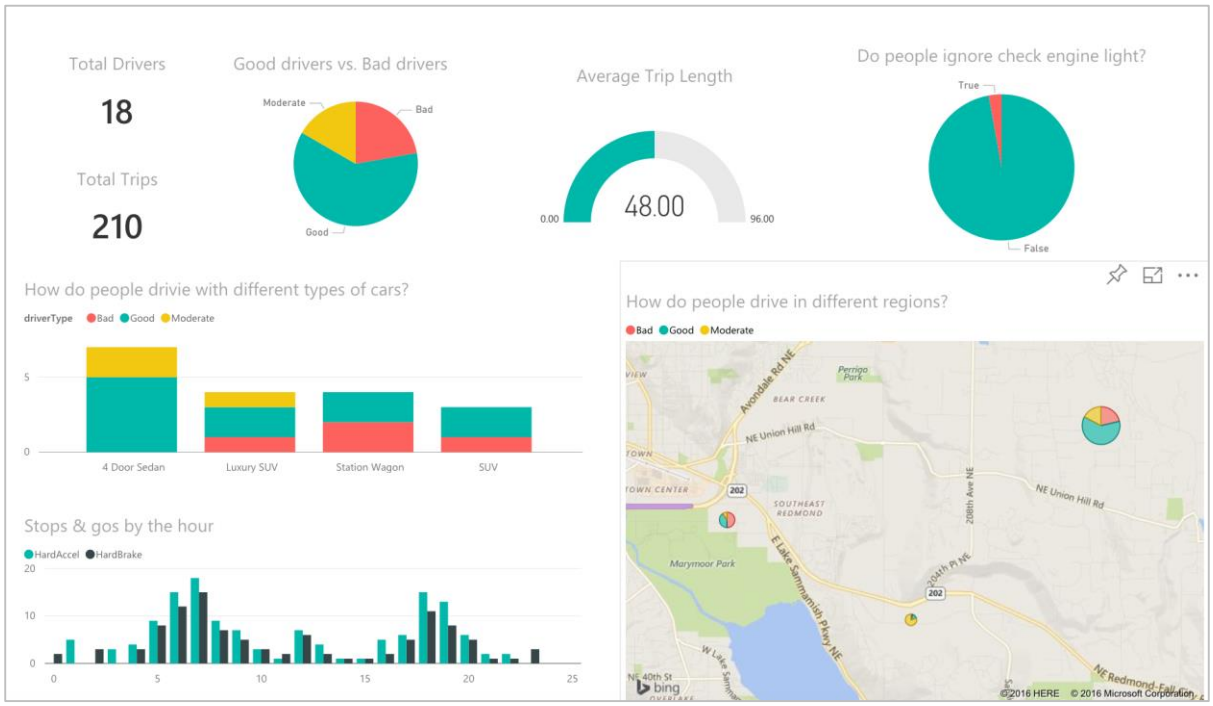


Figure 8-3: The overall system view of MyDriving in Power BI

Microservice Extensions

As we learned in Chapter 6, *Real-Time Data Handling,* Azure Stream Analytics is a service that connects inputs and outputs together through SQL-like processing code. You might also recall that Stream Analytics can connect to a variety of other Azure services on the input and output sides.

Among those other services, Azure Event Hubs is one that can serve in both roles and is also capable of handling high-volume data. This makes it suitable for serving an extensibility role in the MyDriving architecture, along with Azure Service Fabric microservice processing units that are deployed onto Service Fabric clusters, as shown in Figure 9-1.

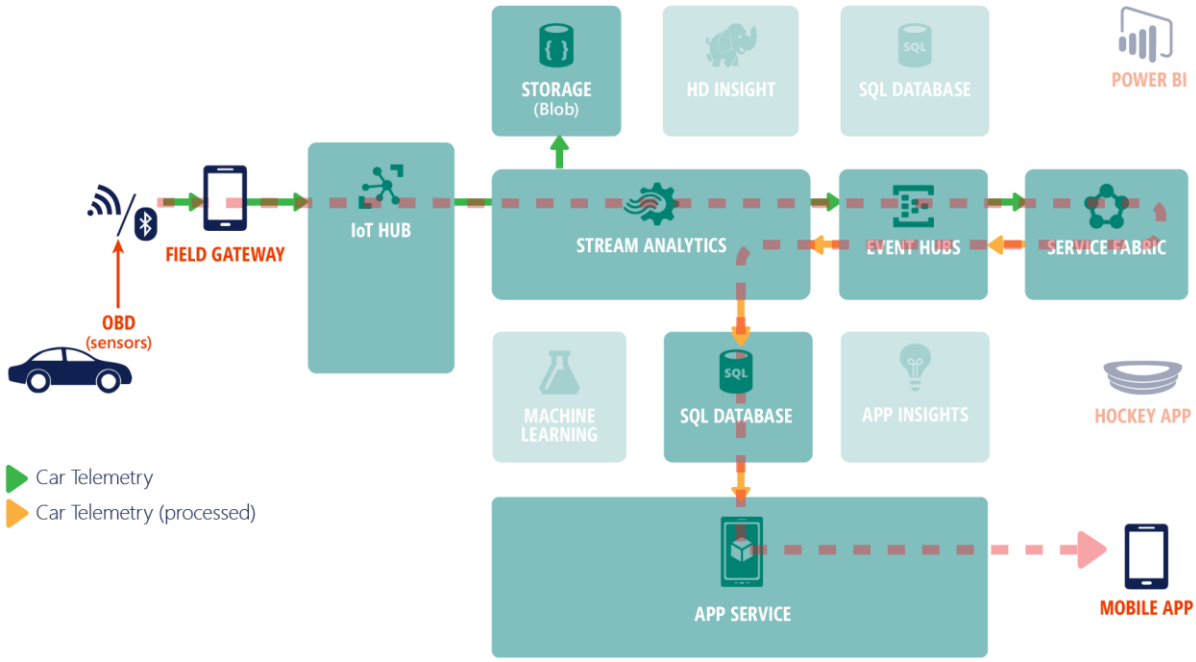


Figure 9-1: Extensibility in MyDriving through Azure Event Hubs and Service Fabric

The flow of data into and through an extension happens as follows:

- A Stream Analytics job pulls data from the Azure IoT Hub input, applies a query, and sends the results to a specific event hub. This is effectively the input for the extension.
- A microservice written in Visual Studio using the Azure Service Fabric SDK and is deployed into a Service Fabric cluster listens to that same event hub. This is how the input that is processed through Stream Analytics gets routed to the code.
- The microservice code performs whatever processing it wants.
- When processing is complete, the microservice routes its results to any number of destinations. For example, it can write the results into Azure storage, or it can send those results to another event hub.
- When you set up another event hub to receive the extension's results, any number of other microservices can listen to that hub and use the data as inputs. A Stream Analytics job can also use the event hub as input, which then routes the microservice output back into Stream Analytics. We'll see some possible patterns at the end of this chapter.

In short, the architecture here is completely open. This allows any number of microservices to be connected however you like—which is simply the nature of Event Hubs and Service Fabric.

In this section, we'll look first at Event Hubs in general. Then, we'll look at how a hub is wired up to receive data from Stream Analytics. We'll then look at Service Fabric and the process of creating and deploying a microservice that listens to the event hub for its input.

Next, we'll walk through the vehicle identification number (VIN) lookup example in the MyDriving solution. This particular example is unidirectional in that it picks up a VIN from incoming on-board diagnostics (OBD) data, looks up additional vehicle information based on that, and writes that data to the SQL database. This makes that information available to consumers, such as the mobile app, through the App Service API endpoints.

Finally, we'll close with a few notes on making connections to other event hubs and microservices, and feeding data back into Stream Analytics.

Primer: Event Hubs

Event Hubs is a highly scalable event-processing service that can ingest millions of events per second. This enables you to process and analyze the massive amounts of data that is produced by connected devices and applications. Event Hubs provides event and telemetry ingestion to the cloud at massive scale, with low latency and high reliability. It also acts as the "front door" for an event pipeline. After data is collected into an event hub, it can be transformed and stored using any real-time analytics provider or batching/storage adapters.

Event Hubs decouples the production of a stream of events from the consumption of those events so that event consumers can access the events on their own schedule. Some other key Event Hubs capabilities are behavior tracking in mobile apps, traffic information from web farms, in-game event capture in console games, and telemetry data that is collected from industrial machines or connected vehicles.

Any entity that sends events or data to an event hub is an *event publisher*. Event publishers can publish events using either HTTPS or AMQP 1.0. Event publishers use a Shared Access Signature (SAS) token to identify themselves to an event hub. They can have a unique identity or can use a common SAS token, depending on the requirements of the scenario.

Entities that read event data from an event hub are *event consumers*, shown in Figure 9-2. All event consumers read the event stream through partitions in a consumer group. The partition only has one active reader at a time. All Event Hubs consumers connect via the AMQP 1.0 session, in which events are delivered as they become available. The client does not need to poll for data availability.

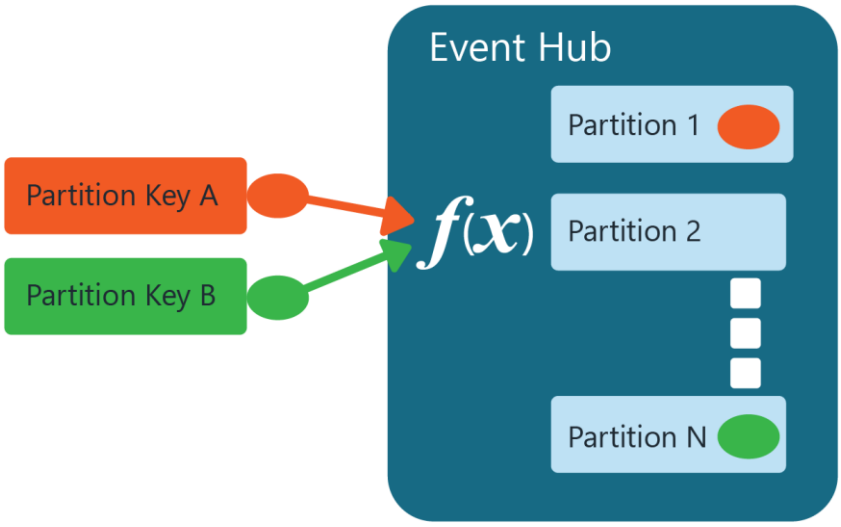


Figure 9-2: Event consumers in Azure Event Hubs

Finally, *consumer groups* (Figure 9-3) enable the publish/subscribe capability of Event Hubs. A consumer group is a view (state, position, or offset) of an entire event hub. Consumer groups enable multiple consuming applications to each have a separate view of the event stream—and to read the stream independently at their own pace and with their own offsets. In a stream-processing architecture, each downstream application equates to a consumer group. If you want to write event data to long-term storage, then that storage writer application is a consumer group. Complex event processing is performed by another, separate consumer group, and so on.

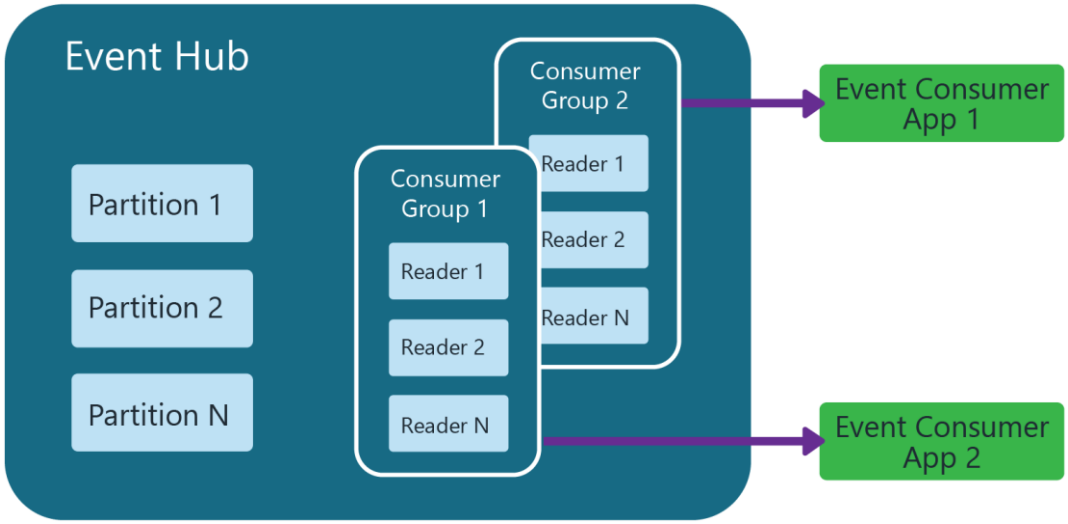


Figure 9-3: Consumer groups in Azure Event Hubs

Applications connect to an event hub using a connection string. Your application can use the connection string to store authentication information and then pass those credentials between

applications. The connection string is created at the Azure Service Bus namespace level and applies to all messaging entities (including Event Hubs) that are created within that namespace.

You obtain a connection string for an Event Hubs namespace (or for any Service Bus namespace) from the Azure classic portal (https://manage.windowsazure.com). When you create a namespace, you use the portal to assign permissions (such as send, receive (listen), or manage) to the connection string. The portal automatically generates a connection string for each of those *shared access policies*.

In the MyDriving solution, the connection string to the Event Hubs namespace (obtained from the Azure classic portal and given **Send** permissions) looks like this:

```
Endpoint=sb://mydriving-ns.servicebus.windows.net/;SharedAccessKeyName=SendData;
SharedAccessKey=<SAS key>
```

The connection string with **Listen** permissions (the receiver) looks very similar:

```
Endpoint=sb://mydriving-ns.servicebus.windows.net/;SharedAccessKeyName=ReceiveData;
SharedAccessKey==<SAS key>
```

Event Hubs in MyDriving: connecting to Stream Analytics

The `mydriving-vinlookup` Stream Analytics job (see Chapter 6, *Real-Time Data Handling,* and also "The VIN lookup extension in MyDriving" later in this chapter) runs a query to extract the VIN number from OBD data, and then it sends that info to Event Hubs for extensibility. This work is performed by the output of the `mydriving-vinlookup` extension, `EHSink`, as shown in Figure 9-4 in the job topology.

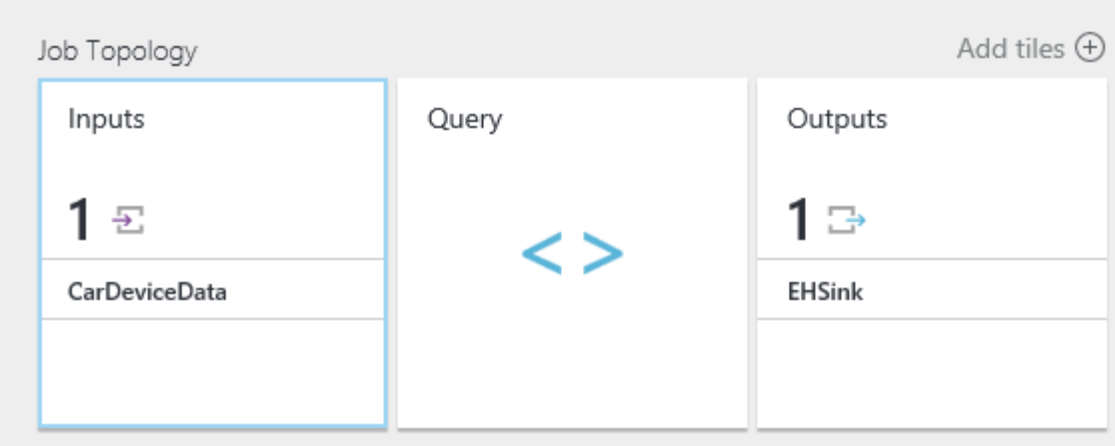


Figure 9-4: Topology of the mydriving-vinlookup job in Stream Analytics

You can click the **Outputs** tab to display the **Outputs** blade, which displays the details of the connection from Stream Analytics to the `mydriving` event hub. To see the details of this event hub, click **Browse** >**Event Hubs**, which opens the Azure classic portal. Click the `mydriving-ns` namespace, click **Event Hubs**, and then click the `mydriving` name. In the dashboard that appears, you can monitor and filter various performance characteristics of the event hub, as shown in Figure 9-5 on the next page.

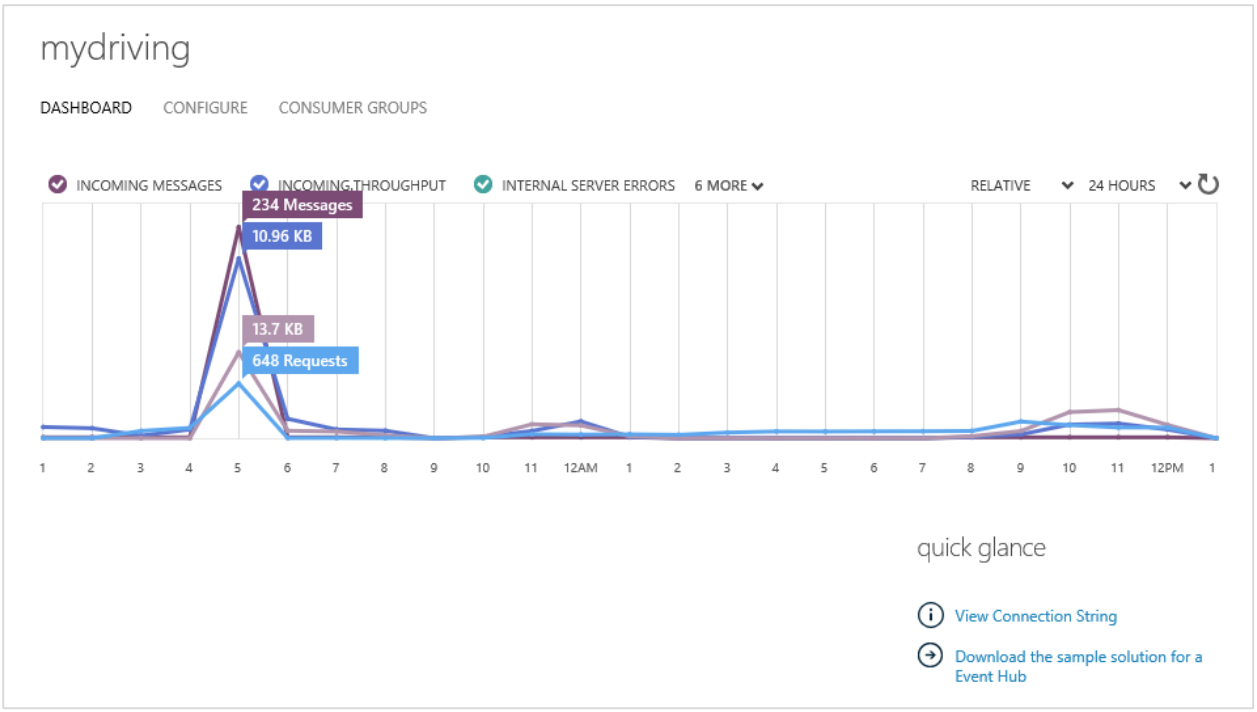


Figure 9-5: The dashboard for the mydriving event hub in the MyDriving system

On this dashboard, you can also access the connection strings that were created for the event hub. Click **View Connection String** to see a dialog box (Figure 9-6) that displays both connection strings, as described earlier in this section. You can also use this dialog box to copy the connection strings to the clipboard, for later use in an application.

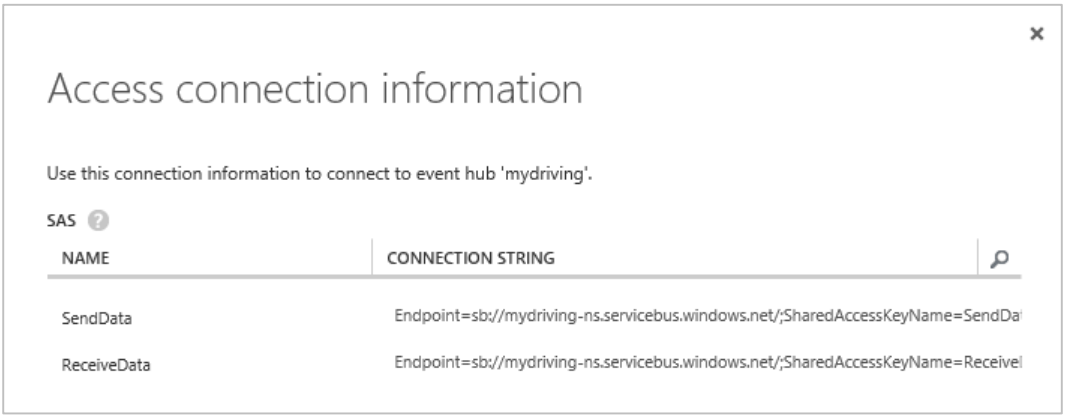


Figure 9-6: Connection strings for the mydriving event hub

To create new shared access policies or regenerate the SAS keys for existing policies, click the **Configure** tab at the top of the portal page as shown in Figure 9-7 (next page).

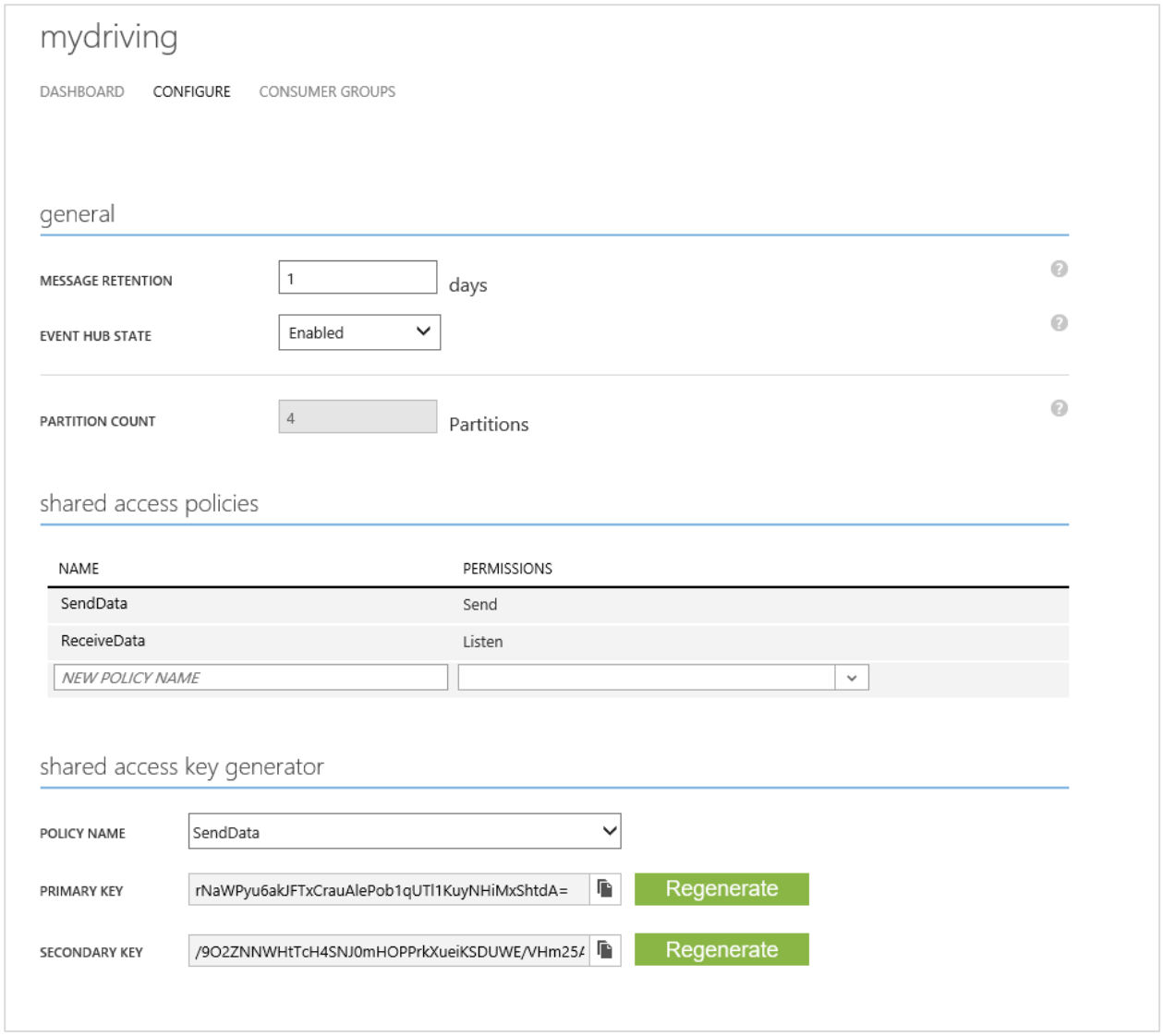


Figure 9-7: Configuring an event hub

Primer: Service Fabric

Azure Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. By using Service Fabric, developers and administrators can avoid the need to solve complex infrastructure problems—and focus instead on implementing mission-critical, demanding workloads, with the knowledge that they are scalable, reliable, and manageable.

Applications composed of microservices

Service Fabric enables you to build and manage scalable and reliable applications that are composed of microservices that run at very high density on a shared pool of machines (referred to as a Service Fabric cluster). Service Fabric also provides comprehensive application management capabilities for provisioning, deploying, monitoring, upgrading/patching, and deleting deployed applications.

The two main advantages of using microservices are as follows:

- They enable you to scale different parts of your application separately, depending on its needs.
- Development teams are able to be more agile in rolling out changes and thereby provide features to your customers faster and more frequently.

Stateless and state-enabled Service Fabric microservices

Stateless microservices (such as protocol gateways and web proxies) do not maintain a mutable state outside of any given request and its response from the service. State-enabled microservices (such as databases and shopping carts) maintain a mutable, authoritative state beyond the request and its response.

There are two .NET Framework programming models that are available for you to build Service Fabric microservices: *Reliable Services* and *Reliable Actors*. The Reliable Services model is a good choice when you need to maintain logic across multiple components and you want to manage service communication. Reliable Actors can be useful when you have lots of independent units of computation/state and you want the platform to manage communication for you.

The VIN lookup extension in the MyDriving solution is composed of a stateless service (implemented using the Reliable Services programming model) and a state-enabled service (implemented using the Reliable Actors programming model).

For more information on application patterns and design, see Service Fabric application scenarios.

Application lifecycle management

Service Fabric provides first-class support for the full application lifecycle management (ALM) of cloud applications—from development through deployment, daily management, and maintenance to eventual decommissioning.

The Service Fabric ALM capabilities enable application administrators/IT operators to use simple, low-touch workflows to provision, deploy, patch, and monitor applications. These built-in workflows greatly reduce the burden on IT operators to keep applications continuously available.

For more information on application lifecycle management, see Service Fabric application lifecycle.

The VIN lookup extension in MyDriving

In MyDriving, the VINLookupApplication processing unit is a Service Fabric application that listens to an event hub, picks out a VIN from incoming OBD data, looks up additional vehicle data based on the VIN, and writes that data to a SQL database.

This application is composed of two services:

- VINLookupService: a stateless service that is implemented using the Reliable Services model
- IoTHubPartitionMap: a stateful service that is implemented using the Reliable Actors model

VINLookupService consumes events from the event hub that is receiving output data from Stream Analytics, picks out the VIN number, looks up additional vehicle information, and saves that information to a SQL database. VINLookupService obtains the event hub partition key from the IoTHubPartitionMap service, which VINLookupService uses to connect to the event hub partition.

We'll take a walk through the VINLookupApplication project and point out the important parts in a moment, but here's a little background information first.

Service Fabric applications

A Service Fabric application is a collection of constituent services. Each service performs a complete and stand-alone function (it can start and run independently of other services)—and is composed of code, configuration, and data. For each service, code consists of the executable binaries, configuration consists of service settings that can be loaded at run time, and data consists of arbitrary static data to be consumed by the service. Each component in this hierarchical application model can be versioned and upgraded independently.

Two different manifest files are used to describe applications and services. The *service manifest* declaratively defines the service type and version. It describes the code, configuration, and data packages that compose a service package to support one or more service types. The *application manifest* declaratively describes the application type and version. It describes elements at the application level and references one or more service manifests to compose an application type. For more information, see Service Fabric application model.

After development on the constituent services is finished, Service Fabric applications are built, packaged, and then deployed onto a Service Fabric cluster. A cluster is a network-connected set of virtual or physical machines into which your microservices are deployed and managed. Clusters can scale to thousands of machines.

You can set up a cluster on Azure through the Azure portal or by using an Azure Resource Manager template (for the latter, see the Resource Manager template for MyDriving on GitHub). In a large solution that uses multiple, related Azure services, it's easier to create and configure the cluster by using a Resource Manager template, rather than doing it manually through the portal. After you have a cluster set up, you're ready to deploy an application. The process of building, packaging, and deploying a Service Fabric application (such as VINLookupApplication) to a local cluster is described in Create your first Azure Service Fabric application in Visual Studio. After you've tested and debugged your application on the local cluster, you can deploy it to a production cluster in Azure using Visual Studio or PowerShell.

VINLookupApplication walkthrough

Now that you know how to set up a cluster and deploy an application, let's take a closer look at the VINLookupService and IoTHubPartitionMap services. Open up the VINLookupApplication.sln solution in Visual Studio and you'll see the VINLookupApplication application project, the VINLookupService and IoTHubPartitionMap service projects, the IoTHubPartitionMap.Interfaces interface project, and the VINParser project, as shown in Figure 9-8.

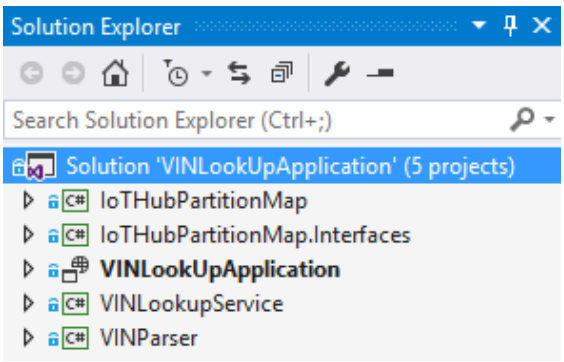


Figure 9-8: Projects in the VINLookUpApplication solution

The application project packages the two services together for deployment and contains the ApplicationManifest.xml and PowerShell scripts for managing the application. The IoTHubPartition-Map.Interfaces project contains the interface definition that is used by the actor in the solution. These service projects define the stateless VINLookupService service and the IoTHubPartitionMap actor. Each service project contains some boilerplate code that does not need to be edited in most cases (Program.cs), the implementation of the service (VINLookupService.cs and IoTHubPartitionMap.cs), as well as configuration settings, data files, and the service manifest file. `VINParser` contains some utility classes for looking up additional vehicle information by VIN.

IoTHubPartitionMap service

The IoTHubPartitionMap service hosts the stateful `IoTHubPartitionMap` actor, which maintains a map of event hub partitions. The PackageRoot/Config/Settings.xml file specifies the event hub connection string and name, which are used by the actor to get the list of partitions for the event hub. Here's a high-level view of the actor, which is defined in IoTHubPartitionMap/IoTHubPartitionMap.cs:

```
internal class IoTHubPartitionMap : StatefulActor<IoTHubPartitionMap.ActorState>,
IIoTHubPartitionMap
{
    IActorTimer _mTimer;
    Task<string> IIoTHubPartitionMap.LeaseTHubPartitionAsync() { /* ... */ }
    Task<string> IIoTHubPartitionMap.RenewIoTHubPartitionLeaseAsync(string partition)
        { /* ... */ }
    protected override Task OnActivateAsync() { /* ... */ }
    private Task CheckLease(Object state) { /* ... */ }
    protected override Task OnDeactivateAsync() { /* ... */ }
    void ResetPartitionNames(){ /* ... */ }

    [DataContract]
    internal sealed class ActorState
    {
        [DataMember]
        public List<string> PartitionNames { get; set; }

        [DataMember]
        public Dictionary<string, DateTime> PartitionLeases { get; set; }
    }
}
```

Actors interact with the rest of the system, including other actors, by passing asynchronous messages using a request-response pattern. These interactions are defined in an interface as asynchronous methods. The `IoTHubPartititionMap` actor type interface is defined in the IoTHubPartitionMap.Interfaces project.

Stateful actors have a state that needs to be preserved across garbage collections and failovers. They derive from the `StatefulActor<TState>`. `TState` (`IoTHubPartitionMap.ActorState` in our example) is the type of state that needs to be preserved, so the actor is preserving a list of event hub partition names and a dictionary of partition leases.

The Reliable Actors runtime automatically activates an actor the first time it receives a request for that actor by calling `OnActivateAsync` and initializing the actor's state. In our example, `OnActivateAsync` reads the event hub's name and connection string from the service configuration settings, creates an `EventHubClient` object, and initializes the actor's `State.PartitionNames` with the list of event hub partitions. A timer is also registered to check the lease on the partitions every 30 seconds.

The `IIoTHubPartitionMap.LeaseTHubPartitionAsync` method is used to lease a partition and obtain a partition name. The `IIoTHubPartitionMap.RenewIoTHubPartitionLeaseAsync` method is used to renew a lease on a partition. A client, such as the VINLookupService service, can invoke these methods through an actor proxy object.

VINLookupService service

The VINLookupService consumes events from the event hub that is receiving output data from Stream Analytics, picks out the VIN number and looks up additional vehicle information, and saves that information to a SQL database. VINLookupService obtains the event hub partition key from the IoTHubPartitionMap service, which the VINLookupService uses to connect to the event hub partition.

The PackageRoot/Config/Settings.xml file specifies the event hub connection string and name, which are used by the service to create an event hub client. This file also specifies a SQL connection string, which is used to create a connection to a SQL database. PackageRoot/Data/ contains additional vehicle data.

The service API provides two entry points for your code: the `RunAsync` and the `CreateService-InstanceListeners` methods. `RunAsync` is where you begin executing workloads, including long-running compute workloads. `CreateServiceInstanceListeners` is a communication entry point where you can plug in your communication stack of choice, such as ASP.NET Web API. VINLookup-Service does not receive requests from users and other services, however, so we'll focus on the `RunAsync` method. The platform calls `RunAsync` when an instance of a service is placed and ready to execute. A cancellation token is provided to coordinate when your service instance needs to be closed.

In the first several lines of `RunAsync`, the event hub connection string, event hub name, and the SQL connection string are read from the PackageRoot/Config/Settings.xml file. An `EventHubClient` object is created and is used to receive events from the event hub. An actor proxy object is also created which the VINLookupService service uses it to interact with the `IoTHubPartitionMap` actor (VINLookupService/ VINLookupService.cs).

```
var configSection = ServiceInitializationParameters.CodePackageActivationContext
    .GetConfigurationPackageObject("Config");

var conStr = configSection.Settings.Sections["ServiceConfigSection"]
    .Parameters["EventHubConnectionString"].Value;

var eventHubName = configSection.Settings.Sections["ServiceConfigSection"]
    .Parameters["EventHubName"].Value;

var sqlConnectionString = configSection.Settings.Sections["ServiceConfigSection"]
    .Parameters["SqlConnectionString"].Value;
```

```
var eventHubClient = EventHubClient.CreateFromConnectionString(conStr, eventHubName);

var proxy = ActorProxy.Create<IIoTHubPartitionMap>(new ActorId(1),
    ServiceInitializationParameters.CodePackageActivationContext.ApplicationName);
```

The additional vehicle data (such as manufacturer, make, and model) that is found in PackageRoot/Data/ is used to create a `VINParser` object. This object will later be used to look up vehicle information by VIN.

```
var dataPackage = ServiceInitializationParameters.CodePackageActivationContext
    .GetDataPackageObject("Data");

VINParser.VINParser parser = new VINParser.VINParser(
    Path.Combine(dataPackage.Path, "wmis.json"));
```

At this point we jump right into a `while` loop, which runs our workload, so there is no need to schedule a separate task for the workload. The workload runs until the system requests cancellation (when the service is shut down). An event hub partition name is obtained from the actor proxy, which is then used to create an `EventHubReceiver` object. A second `while` loop is used to continuously receive event hub event data. A VIN number is extracted from the received event data, the `VINParser` retrieves any additional vehicle information, and that vehicle information is persisted to a SQL database.

```
while (!cancelServiceInstance.IsCancellationRequested)
{
    string partition = proxy.LeaseTHubPartitionAsync().Result;
    if (partition == "")
        await Task.Delay(TimeSpan.FromSeconds(15), cancelServiceInstance);
    else
    {
        var eventHubReceiver = eventHubClient.GetDefaultConsumerGroup()
            .CreateReceiver(partition, DateTime.UtcNow);
        while (!cancelServiceInstance.IsCancellationRequested)
        {
            EventData eventData = await eventHubReceiver.ReceiveAsync();
            if (eventData != null)
            {
                string data = Encoding.UTF8.GetString(eventData.GetBytes());
                TripVIN vin = JsonConvert.DeserializeObject<TripVIN>(data);
                try
                {
                    var carInfo = parser.Parse(vin.VIN);
                    SaveRecord(sqlConnectionString, vin.VIN, carInfo);
                }
                catch (Exception exp)
                {
                    ServiceEventSource.Current
                        .ServiceRequestFailed("VINLookupService",
                               "Failed to Save VIN: " + exp.Message);
                }
            }
            if (DateTime.Now - timeStamp > TimeSpan.FromSeconds(20))
            {
                string lease = proxy.RenewIoTHubPartitionLeaseAsync(partition).Result;
                if (lease == "")
                    break;
            }
        }
    }
}
```

Additional extension routes

The VIN lookup extension that is included with MyDriving is but one of a number of extensions that you could build. Here are some other examples:

- A microservice could take the GPS coordinates from the car telemetry and use them to identify nearby points of interest, such as restaurants or museums. The microservice could feed these suggestions into the real-time data flow to include in information that is displayed in the mobile app.
- A solution similar to MyDriving might be used in the management of a fleet of delivery vehicles to monitor drivers. A microservice could take the VIN and cross-reference it with another line-of-business system to determine the identity of the driver who is driving the vehicle at that time. The microservice could write this data to another event hub that feeds back into the Stream Analytics job. From there, the driver identity can be included in the data that is written to cold storage.

These examples show that extensions aren't limited to a one-way path from Stream Analytics to a microservice and into storage, as with the VIN information. Stream Analytics, if you remember from Chapter 6, can use Event Hubs as either an input or an output. It's entirely possible then to string together any number of hubs with different microservices that route data through many operations—and then feed it back to Stream Analytics. A few of these possibilities are illustrated in Figure 9-9.

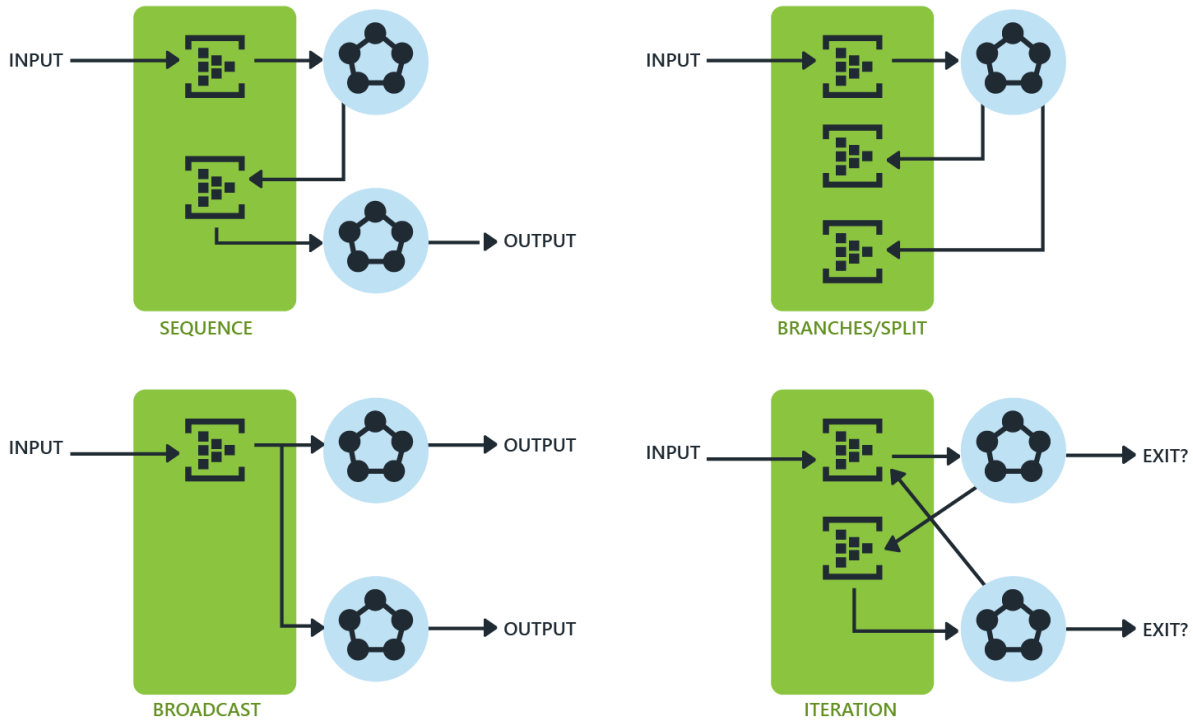


Figure 9-9: Other possible extension patterns using Event Hubs and Service Fabric

Reference

OBD data schema

These are the values that the OBD libraries used in the mobile app (acting as a field gateway) poll from the OBD device.

See OBD-II Parameter IDs for additional details. Some fields might be empty if a vehicle doesn't support them.

PID	Name	Units
04	Engine load	%
06	Short-term fuel % trim - Bank 1	%
07	Long-term fuel % trim - Bank 1	%
0C	Engine RPM	rpm
0D	Vehicle speed	km/h
10	MAF air flow rate	grams/second
11	Throttle position	%
1F	Run time since engine start	seconds
21	Distance traveled with malfunction indicator light on	km
45	Relative throttle position	%
46	Ambient air temperature	°C
5E	Engine fuel rate	liters/hour

Application data schema

These are the data objects used throughout the back end and the MyDriving mobile app. See the MyDriving.DataObjects project in the MyDriving app solution for the class definitions.

IOTHubData

Represents data that is collected from an IoT device as it's sent to IoT Hub.

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)

Blob	string	The IoT data in JSON, formatted as follows (the tripId and userId come from the current Trip being recorded):
		`{`
		`  {`
		`  "TripId" : <tripID>,`
		`  "UserId" : <userID>`
		`  },`
		`  "TripDataPoint" : {`
		`  <serialized TripPoint object>`
		`  }`
		`}`

Photo

Represents a photo that is taken while recording a trip (not presently used in the MyDriving app).

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)
TripId	string	The identifier of the associated Trip
PhotoUrl	string	The URL to the stored photo
Latitude	double	The latitude at which the photo was taken
Longitude	double	The longitude at which the photo was taken
TimeStamp	DateTime	The time at which the photo was taken

POI (points of interest)

Represents a point during a trip when a hard stop or hard acceleration was detected.

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)
TripId	string	The identifier of the associated Trip
Latitude	double	The latitude at which the event occurred
Longitude	double	The longitude at which the even occurred
POIType	POIType	A value from the POIType enumeration (see the following table) that indicates the type of event
TimeStamp	DateTime	The time at which the event occurred

POIType (enum)

Name	Value
HardAcceleration	1
HardBrake	2

Trip

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)
Name	string	The trip name, as entered by the user at the end of recording
UserId	string	The user ID (GUID) for the authenticated user, generated by the back end
Points	List<TripPoint>	A collection of **TripPoint** objects that describe the route
RecordedTimeStamp	DateTime	The time the trip recording started
EndTimeStamp	DateTime	The time the trip recording stopped
Rating	int	Not used
IsComplete	bool	The flag that indicates if the trip is complete (true) or still being recorded (false)
HasSimulatedOBDData	bool	The flag that indicates if simulated OBD data was used for this trip (true) or if real data was recorded (false)
AverageSpeed	double	The average speed for the trip
FuelUsed	double	The fuel usage for the trip

HardStops	long	The number of hard stops during the trip
HardAccelerations	long	The number of hard accelerations during the trip
MainPhotoUrl	string	The URL of the main photo that is associated with the trip
Distance	double	The total distance that is covered during the trip

TripPoint

For data coming from OBD, see OBD-II Parameter IDs for more details.

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)
TripId	string	The unique identifier of the parent Trip
Latitude	double	The latitude of the point (as reported by GPS)
Longitude	double	The longitude of the point (as reported by GPS)
Speed	double	[OBD] The current speed in km/h
RecordedTimeStamp	DateTime	The time that the point was recorded
Sequence	int	The sequence number for the point
RPM	double	[OBD] The current engine RPM
ShortTermFuelBank	double	[OBD] The current short-term fuel trim %
LongTermFuelBank	double	[OBD] The current long-term fuel trim %
ThrottlePosition	double	[OBD] The current throttle position %
RelativeThrottlePosition	double	[OBD] The current relative throttle position %
Runtime	double	[OBD] The length of time in seconds that the engine has been running
DistanceWithMalfunctionLight	double	[OBD] The distance that has been traveled since the malfunction indicator light switched on, in km
EngineLoad	double	[OBD] The current engine load %
MassFlowRate	double	[OBD] The current air flow rate from the mass air flow sensor in grams/second
OutsideTemperature	double	[OBD] The external (ambient air) temperature in °C
EngineFuelRate	double	[OBD] The current engine fuel rate in liters/hour
VIN	string	[OBD] The unique 17-character vehicle identification number (VIN) (with identity bits removed)
HasOBDData	bool	The flag that indicates whether the trip point has OBD data

HasSimulatedOBDData	bool	The flag that indicates if simulated OBD data was used for this trip (true) or if real data was recorded (false)

UserProfile

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)
FirstName	string	The user's first name
LastName	string	The user's last name
UserId	string	The user ID (GUID) for the authenticated user
ProfilePictureUri	string	The URI to the user's profile picture (often taken from identity providers like Facebook or Twitter)
Rating	int	The overall driver rating across all trips
Ranking	int	Not used
TotalDistance	double	The total distance that this user has traveled, in miles
TotalTrips	long	The total number of trips taken by this user
TotalTime	long	The total driving time for this user
HardStops	long	The total number of hard stops for this user
HardAccelerations	long	The total number of hard accelerations for this user
FuelConsumption	double	The total fuel consumption for this user, in gallons
MaxSpeed	double	The maximum speed for this user across all trips, in km/h
Device	List<Device>	A collection of the user's registered IoT devices (see the following table for Device type)

Device

Property	Type	Description
Id	string	A unique identifier for the object (GUID, inherited from BaseDataObject)
Name	string	The name of the device

App Service tables

Unless otherwise noted for a specific table, endpoints for tables follow the pattern in the following table, where the name of the table in <*table*> is case insensitive. Note that the information that is accessible via tables is always scoped to the currently authenticated user and requires authentication with the back end.

Operation	Retrieve single object, retrieve all objects
HTTP Method	GET
URI	Retrieve all: /tables/<table> Retrieve single object: /tables/<table>/<id> URIs can include standard oData queries
Headers	ZUMO-API-VERSION=2.0.0
Returns	Single object or array of objects in JSON

Operation	Add object
HTTP Method	POST
URI	/tables/<table>
Headers	ZUMO-API-VERSION=2.0.0; Content-Type=application/json
Payload	JSON object to insert
Returns	HTTP 200 with inserted object

Operation	Update the object identified by <id>
HTTP Method	PATCH
URI	/tables/<table>/<id>
Headers	ZUMO-API-VERSION=2.0.0; Content-Type=application/json
Payload	JSON that contains partial object JSON with fields to update
Returns	HTTP 200 with updated object

Operation	Delete object identified with <id>
HTTP Method	DELETE
URI	/tables/<table>/<id>
Headers	ZUMO-API-VERSION=2.0.0
Returns	HTTP 204

IoTHubData

Supports all operations.

POI

Supports only GET for all POIs for a trip by using either of the following URIs:

```
/tables/POI?tripId=<tripId>
```

```
/tables/POI(<tripId>)
```

Trip

Supports all operations. Callers can also query directly into the Point collection using the URI */tables/trip/points*.

TripPoint

Supports all operations, but callers will typically want to filter trip points with a query. For example, the following returns the next 10 trip points from trip '2' starting from sequence number 11:

```
$filter=tripId%20eq%20'2'%20and%20sequence%20gt%2011%20and%20IsEvent%20eq%20true&count=10
```

UserProfile

Supports all operations. Callers can also query directly into the Device collection using the URI */tables/trip/devices*.

App Service APIs

These are the additional non-table APIs that are implemented in App Service in the back end.

Provision

Operation	Provision a device
HTTP Method	GET
URL	/api/provision?userId=<userId>&deviceName=<device_name> Each user is allowed to register up to three unique device names. If the limit is exceeded, the user receives a 400 Bad Request response.
Headers	ZUMO-API-VERSION=2.0.0
Returns	Device access key (string)

UserInfo

Operation	Retrieve information for the currently authenticated user
HTTP Method	GET
URL	/api/userinfo
Headers	ZUMO-API-VERSION=2.0.0
Returns	The UserProfile object for the current user

Azure SQL Database schema

These are additional tables that are defined by sql_createtables.txt, as used by the Azure Resource Manager template.

dimUser

Associates a userId with a VIN.

Field	Type	Description
userId	nvarchar(100)	A unique identifier for the user (GUID)
vinNum	nvarchar(20)	The VIN for the user's current vehicle

dimVinLookup

Describes the additional data that is retrieved by the microservice extension by using a VIN.

Field	Type	Description
vinNum	nvarchar(20)	The VIN for the vehicle
make	nvarchar(200)	The make of the vehicle
model	nvarchar(200)	The model of the vehicle
carYear	int	The release year of the vehicle
carType	nvarchar(200)	The type of the vehicle

factTripData

Used to store results from Azure HDInsight for use in training the machine learning module.

Field	Type	Description
userId	nvarchar(100)	A unique identifier for the user (GUID)
tripId	nvarchar(200)	The trip identifier (GUID)
driverType	nvarchar(10)	Descriptive text for the driver rating
avgSpeed	float	The average speed for the trip
numOfHardAccel	int	The number of hard accelerations for the trip
numOfHardBrakes	int	The number of hard brakes for the trip
emissionLevel	float	The level of emissions for the trip
wasMILOn	bit	The flag that indicates whether the indicator light was on during the trip
timeOfTrip	time(7)	The total time of the trip
cLatitude	float	The average latitude during the trip
cLongitude	float	The average longitude during the trip

Azure Stream Analytics query examples

Filter by adding a WHERE clause:

```
SELECT *
INTO OutputStream1
INPUT InputStream1
WHERE dspl='sensorA'
```

Rename and project fields:

```
SELECT
  time,
  dspl AS SensorName
  temp AS Temperature
 INTO OutputStream1
 INPUT InputStream1
 WHERE dspl='sensorA'
```

Moving average: *TIMESTAMP* identifies the *time* field as the time over which to perform the tumbling window average. *TumblingWindow* groups the records into successive intervals of the given period.

```
SELECT
  System.Timestamp AS OutputTime,
  dspl AS SensorName,
  Avg(temp) AS AvgTemp
INTO
  OutputStream1
FROM
  InputStream1 TIMESTAMP By time
GROUP BY TumblingWindow(second, 60), dspl
```

Self-join: This query creates two aliases, *t1* and *t2,* for a single input stream and performs a join between the two. Where there is a record from a particular sensor that is not matched by another record within 60 seconds, the query generates an output row. The query therefore generates an output when no data has been received from a previously live sensor for the past minute.

```
SELECT
  t1.time,
  t1.dspl AS SensorName
INTO
  output
FROM
  InputStream1 t1 TIMESTAMP BY time
LEFT OUTER JOIN
  InputStream1 t2 TIMESTAMP BY time
ON
  t1.dspl=t2.dspl AND
  DATEDIFF(second, t1, t2) BETWEEN 1 and 60
WHERE t2.dspl IS NULL
```

Build and Release configurations in Visual Studio Team Services

We use the cloud build, test, and release management service of Visual Studio Team Services. It does much of the build and test, but we use Xamarin Test Cloud for automated testing of mobile apps. We use HockeyApp for distribution. Source is kept in GitHub. All of these services are integrated into Team Services through extensions that are available in the Marketplace, where there are also other useful utilities.

We use continuous integration so that every code check-in triggers a build and test.

MyDriving.Services build definition

Build step	Parameters	Purpose
NuGet Installer 	src/MobileAppService/MyDrivingService.sln	Restore packages so that we don't have to keep them on repo
Visual Studio Build 	MobileApp/smarttrips.sln /p:DeployOnBuild=true /p:WebPublishMethod=Package /p:PackageAsSingleFile=true /p:SkipInvalidConfigurations=true /p:PackageLocation="$(build.artifactstagingdirectory)\\"	Build solution for mobile app back end
Visual Studio test 	**\$(BuildConfiguration)*test*.dll;-:**\obj**	Run tests
Copy files 	$(build.sourcesdirectory) **\bin\$(BuildConfiguration)** $(build.artifactstagingdirectory)	
Publish build artifacts 	$(build.artifactstagingdirectory)	Drop to the web server

Mydriving.Xamarin.Android build definition

Build step	Parameters	Purpose
NuGet Installer 	src/MobileApps/MyDrivingXS.sln	Restore packages so that we don't have to keep them on repo

Version Assemblies 	src/MobileApps/MyDriving/MyDriving.Android/Properties AndroidManifest.xml (?:\d+\.\d+\.\d+\.)(\d+)	Update build version name
Version Assemblies 	src/MobileApps/MyDriving/MyDriving.Android/Properties AndroidManifest.xml (?:\d+\.\d+\.\d+\.)(\d+)	Update build version code
cmd line 	curl -k "https://....blob.core.windows.net /buildassets/smarttrips.keystore?...= " --output $(Build.SourcesDirectory)\src\MobileApps\ MyDriving\MyDriving.Android\smarttrips.keystore	Download keystore—keep keys in storage instead of in code
Xamarin License 	Activate Xamarin license $(XamarinLicenseUser)	Activate license on this build VM
Xamarin Android Build 	src/MobileApps/MyDriving/ MyDriving.Android/MyDriving.Android.csproj	Build project
Android Signing 	$(Agent.BuildDirectory)\bin\$(BuildConfiguration)*.apk $(Build.SourcesDirectory)\src\MobileApps\ MyDriving\MyDriving.Android\smarttrips.keystore -verbose -sigalg MD5withRSA -digestalg SHA1	Sign and align the .apk file
Xamarin License 	Deactivate license	Remove it from this build VM so it can be used on another
MSBuild 	src/MobileApps/MyDriving/ MyDriving.UITests/MyDriving.UITests.csproj	Build tests
Xamarin Test Cloud 	$(Agent.BuildDirectory)\bin\$(BuildConfiguration) \com.microsoft.mytrips.apk	Run tests in cloud test service
Copy files 	$(build.sourcesdirectory) **\bin\$(BuildConfiguration)** $(build.artifactstagingdirectory)	
Publish build artifacts 	$(build.artifactstagingdirectory)	Drop to the web server

Mydriving.Xamarin.iOS build definition

Build step	Parameters	Purpose
Cmd 	rm ../../../../.xamarin-credentials	Remove Xamarin credentials
Cmd 	mono $(Build.SourcesDirectory)/src/MobileApps/tools/xamarin-component.exe login $(XamarinLicenseUser) -p $(GeneralPassword)	Add Xamarin credentials
Cmd 	mono $(Build.SourcesDirectory)/src/MobileApps/tools/xamarin-component.exe restore $(Build.SourcesDirectory)/src/MobileApps/MyDriving.iOS.sln	Restore Xamarin components
Cmd 	/usr/local/bin/nuget restore $(Build.SourcesDirectory)/src/MobileApps/MyDriving.iOS.sln	Restore NuGet packages
Xamarin iOS Build 	src/MobileApps/MyDriving.iOS.sln	Build project
Xcode Package iOS 	MyDrivingiOS.app MyDrivingiOS.ipa	Package app
Copy files 	$(build.sourcesdirectory) **\bin\$(BuildConfiguration)** $(build.artifactstagingdirectory)	
Publish build artifacts 	$(build.artifactstagingdirectory)	Drop to the web server

Mydriving.Xamarin.UWP build definition

Build step	Parameters	Purpose
NuGet Installer 	src/MobileApps/MyDriving.sln	Restore packages so that we don't have to keep them on repo
Cmd 	curl -k "https://....blob.core.windows.net /buildassets/MyTrips.UWP_TemporaryKey.pfx?... --output $(Build.SourcesDirectory)\src\MobileApps\ MyDriving\MyDriving.UWP\MyDriving.UWP_TemporaryKey.pfx	Download keystore—keep keys in storage instead of in code
Xamarin License 	Activate Xamarin license $(XamarinLicenseUser)	Activate license on this build VM
Version assemblies 	src/MobileApps/MyDriving/MyDriving.UWP/Properties AssemblyInfo.* \d+\.\d+\.\d+	Update build version name
Version assemblies 	src/MobileApps/MyDriving/MyDriving.UWP Package.appxmanifest \d+\.\d+\.\d+\.\d+	Update build version code
MSBuild 	src/MobileApps/MyDriving/ MyDriving.UWP/MyDriving.UWP.csproj $(BuildConfiguration) /p:AppxBundlePlatforms="$(BuildPlatform)" /p:AppxPackageDir="$(Build.BinariesDirectory)\AppxPackages\\" /p:AppxBundle=Always /p:UapAppxPackageBuildMode=CI	Build solution
Xamarin License 	Deactivate license	Remove it from this build VM so it can be used on another VM
Copy files 	$(Build.BinariesDirectory)\AppxPackages ***$(Build.BuildNumber)*.appxupload $(build.artifactstagingdirectory)	
Publish build artifacts 	$(build.artifactstagingdirectory)	Drop to the web server

MyDriving.Services release definition

Task	Parameters	Purpose
Azure web app deployment	$(System.DefaultWorkingDirectory)***.zip -connectionString @{"$(ConnectionStringName)"= "Server=tcp:$(ServerName).database.windows.net,1433; Database=$(DatabaseName);User ID=$(AdministratorLogin)@$(ServerName); Password=$(AdministratorLoginPassword); Trusted_Connection=False;Encrypt=True;"}	Deploy to the web server
Visual Studio test 	***test*.dll;-:**\obj**	Run tests

MyDriving.Xamarin.Android, iOS, and UWP

Similar task for each:

HockeyApp deployment 	Build HockeyApp $(HockeyAppId) $(System.DefaultWorkingDirectory)\ MyDriving.Xamarin..../drop/bin/ $(Build.BuildNumber)/com.microsoft.mydriving....	Deploy to HockeyApp for distribution