O'REILLY®

Compliments of GARMIN®

Wearable Programming for the Active Lifestyle

Using Garmin Connect IQ™

Brian Jepson

GARMIN

Develop Apps for Garmin Devices

Set your innovation in motion by developing apps for Connect IQ™. Our platform is continuing to expand, so now's your chance to build the next great app and get your work in front of a massive global audience.

- **Build your brand**
- **Enhance your solution**
- **Generate revenue**

Download the SDK

works with GARMIN connect IQ

Wearable Programming for the Active Lifestyle

Using Garmin Connect IQ

Brian Jepson

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Wearable Programming for the Active Lifestyle
by Brian Jepson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Jepson and Jeff Bleiel
Production Editor: Melanie Yarbrough
Copyeditor: Jasmine Kwityn
Proofreader: Sonia Saruba

Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

November 2016: First Edition

Revision History for the First Edition
2016-11-04: First Release

978-1-491-97207-6

[LSI]

Table of Contents

Preface

As computing devices have gotten smaller and smaller, and as they have gotten more interested in knowing things about you, they have also become more intimate. A desktop computer that you control with a keyboard and mouse doesn't feel like an extension of you most of the time (with a notable exception being when you're lost in an activity such as programming or gaming). A tablet feels an awful lot like a book. Mobile phones get a little closer to you, but they still have enough bulk to constantly remind you they are still there.

Wearable devices, however, can truly begin to feel like a part of you. You might use them to tell time and receive notifications from your mobile phone, but you also use them in a much more personal fashion. They can monitor your heart rate and count your steps, and when they need your attention, they touch you gently by vibrating.

Balancing Comfort, Looks, and Ability

Over the years, we've attached some clunky-looking devices to our wrists, our belts, and even our eyeglasses. Some smart jewelry can hang from your neck or even ears. But when does a device cease to be an appendage and start feeling like an extension of yourself? It's probably a device that you forget to take off when you go for a swim, that you can wear to the top of a mountain and back down, and that you can comfortably wear while sleeping.

A smartwatch works in all of those use cases. It's the least obtrusive thing that can possibly work. A few hours into your swim, hike, or nap, and it's an extension of you that's always there.

A smartwatch can be the center of your personal area network. To succeed at it, it's got to not only be a seamless extension of your person, but it's also got to have a design you want to look at continuously, and it's got to do something, and do it well.

How Big, How Hungry?

Given this intimacy of the smartwatch, it's no wonder that reviewers and users alike fret over the short battery life that many devices have. If you think of your wearable as nothing more than an extension of your smartphone, maybe it's not unreasonable for it to have a charge that lasts one or two days. But if it's an extension of you, you need to be able to trust the device…wherever you are, whatever you do, and whenever you need it.

For a device to be a constant companion, something that you can take into the wild for a week or on a cross-country bike trip, you need days of battery life. After all, you're expecting your device to observe and record not only your heart rate or step count but the geographic points you travel over in your wanderings. Whatever you do, you need the device and its accessories to last through your activities, and then some.

It's not all about the battery, but many other factors. If you're assuming you'll use the device outdoors, you need to be able to read its screen outdoors. The device can't be so big as to be clunky (and neither can its peripherals). It needs to be easy to interact with, it needs to look good, and it needs to stand up to the elements, slips, falls, and impacts.

What Do You Want?

A wearable device becomes a trusted and natural part of our lives. A good wearable will have features and behaviors that come from human needs. Humans move around, engage in activities, and occasionally need to be reminded to not sit still. As a companion and extension of us, the device needs to know what we are doing. Are we sitting still or in motion? How fast are our hearts beating?

It needs to know these things, and it needs to know them 24/7. And it needs to be connected. It's one thing to gather this data and display it. It's another to allow users to connect their data to apps and systems that are part of a larger health and wellness ecosystem. For

all this to come together, developers like you will need to not only understand how to develop for these devices, but how to develop for low-power, resource-constrained scenarios. A device manufacturer can set the stage with power-sipping devices, but app developers need to stick to the game plan to keep from overtaxing the device's battery.

Connections

Aside from the question of what you expect a device to do, there is also the matter of how it does it. A wearable can't do everything, so it needs to rely on devices around it. As a result, it needs to make connections to other devices: a smartphone is a hub that connects the wearable to the cloud.

But what will it say to the hub? It's tempting to imagine a hub that's just a gateway, allowing the wearable device to make direct connections over whatever network the hub is connected to. But that assumes the wearable has a lot of memory and computational power. Memory and computational power are relatively cheap, but for any work they do, there is a corresponding power draw. Once again, with a constraint comes an opportunity to define a better interaction model.

There are a few key interactions that you'll engage in when you connect to the world beyond your wrist:

Phone notifications
: As much as possible, you will want your users to interact with your apps directly through the wearable device. But there will be times, such as when a user needs to provide credentials to connect a web app to your app, that you'll need to direct the user's attention to his phone to complete a task.

Interacting with the Internet
: A Garmin wearable can interact with web apps, typically through an API. When this happens, interactions need to be quick and brief. There's no reason to pull down 16 kilobytes of JSON when you only need a 16-byte string from it. Save memory, save power.

Be part of the IoT
: Users are surrounded by other smart devices, from sensors to entertainment devices to smart homes. Through the use of

APIs, and also through short-range sensor technology like ANT+, your device can communicate with any IoT device that exposes an API or offers the ability to connect wirelessly.

The Platforms

Power consumption is one key area where smart device platforms show their subtle differences. One one end, the power-hungry end, you've got Android Wear and Apple Watch, with battery life just over a day. The Pebble smartwatches trade battery life (2 to 10 days depending on model) for a slower CPU without really sacrificing functionality. Garmin devices can run even longer—two to three weeks (between 8 and 14 days with 24/7 heart rate monitoring).

What the Pebble and Garmin wearables lack are actually their strengths. Because they choose reflective displays that are readable in sunlight, they are usable where you probably use them most: outdoors. And although they don't have CPUs as fast as the Apple Watch or Android Wear devices, they get great battery life, and still have enough built-in computational power to create a user experience that works well on the small screen.

Of all the smartwatches on the market, Garmin devices have been designed for the most extreme of conditions. An ultramarathoner who needs to track activity for a 100 km footrace may expose the device to varying weather conditions, needs the device to be readable in bright sunlight, and needs it to stand up to 12 or more hours of sweat and strain. And while the user and the elements are trying to destroy it, the device needs to log position, speed, heart rate, and maybe more…every second of those 12 hours.

Inventing the Future of Wearable Devices

In these modern times, where computers and mobile devices have massive amounts of memory, computing power, and battery life, wearables represent a multifold challenge. First, you need to work within significant constraints in the face of modern user expectations. Second, you're going to find yourself at the leading edge of a new kind of development; with that comes the opportunity to blaze a trail, but with the challenge of finding your own way. Finally, you're inventing the future; as exciting as that is, it does come with a great responsibility to get it right.

With all the wearable platforms out there, why Garmin devices? First of all, they are built to stand up to grueling conditions, have long battery lives, and there are millions of devices on the wrists of millions of dedicated, active fans of the Garmin brand.

In this book, you'll see some of the many things you can do with the Garmin Connect IQ platform. The device APIs let you program to the various capabilities of the wearable device. ANT and ANT+ wireless communication expands your solutions to include external sensors—from off-the-shelf to something you create. And with support for authenticating to web services in Connect IQ, your data won't be trapped on the device.

Acknowledgments

When I first began working on this book, I was in way over my head. Fortunately, I had plenty of curiosity and several immensely helpful guides to help me on my way: a big thanks to Josh Gunkel, Nick Kral, and Nate Ahuna for helping me understand the Garmin device family as well as help me find my way around the Connect IQ SDK. Thanks also to Sebastian Barnowski and Harrison Chin for helping me learn more about ANT and ANT+, and to Laura McClernon for helping me understand the Garmin wellness programs.

CHAPTER 1
Big Data and µData

Years ago, you could use a conventional database system to store, process, and display pretty much any kind of data you might come across. These days, thanks to ever-present sensors and the ability to obtain large amounts of information in real time, our data has gotten too big, and it changes shape almost as fast as it accumulates.

Whether it's data from high-speed stock market trades or information streaming in from a heart rate monitor, it's big and hard to control. *Big data* has emerged as the catch-all term for both the data itself and also for the tools and practices we use to get it under control.

These tools and practices give us a better understanding of the data through more efficient and more enlightening analysis. Applied to financial data, it might make some of us richer. But applied to health and fitness, we can use big data techniques to help live longer, healthier lives.

The *quantified self* movement uses technology to capture data about as many aspects of human life as can be measured. Even a single individual can generate an incredible amount of data, depending on what you're monitoring. Every dimension you add—heart rate, blood pressure, blood oxygen level—gets projected over time, so if you're monitoring 24/7 and sampling every second, the amount of data gets huge.

Armed with the right devices and software, you can measure yourself, gain insights that would not be otherwise possible, and make your life better.

The Garmin "Full Circle Experience"

The Garmin "Full Circle Experience" (Figure 1-1) defines all the parts of its ecosystem that work together to create one experience that is wholly driven by a user's fitness and wellness data stream.

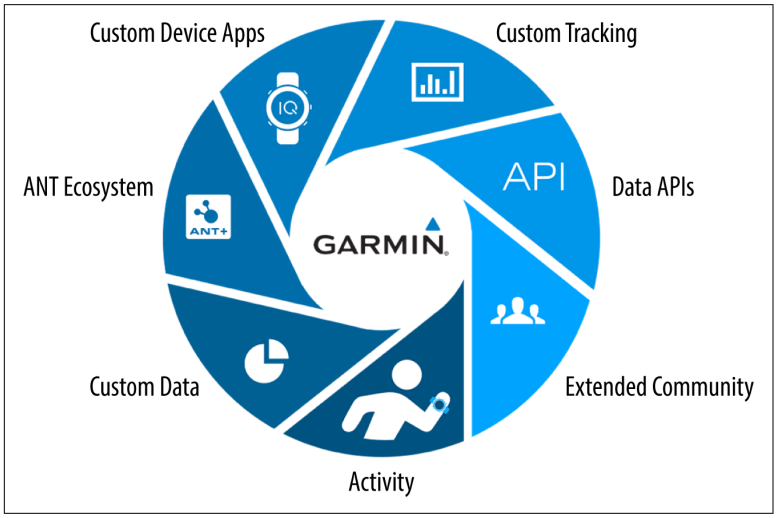

Figure 1-1. Garmin's Full Circle Experience

Let's briefly discuss each part of it:

Activity
: At the bottom of the circle, you can see the user engaged in an activity. This is where it begins. The data that the wearable measures and collects is all tied to whatever activity the user is engaged in at that moment. Your job as a developer is to organize that information in a way that makes sense for a given activity, and bring it to the user in a relevant way.

  Out of the box, a Garmin device may include support for many activities: running, cycling, walking, swimming, golfing, rowing, stand-up paddleboarding, skiing, and much more.

Custom Data

Every activity that your device is capable of tracking has data fields associated with it. For example, there are several data fields associated with the heart rate monitor: current heart rate, average heart rate, heart rate zone (1-5, with zones based on user profile factors such as age), and others. These fields can be displayed on an activity screen on your device, but they can also be exported to applications and APIs in the FIT format.

As a developer, you can add your own custom data to this experience with Connect IQ data fields. For example, the Strava social network for athletes has its own "Strava Live Suffer Score," a custom data field that analyzes your workout data to tell you how hard you're trying.

ANT Ecosystem

When the user starts an activity, the device begins recording data. That data can come from an onboard sensor, such as a built-in heart rate sensor, or from a wireless sensor the user is wearing (or has attached to a bike or exercise equipment).

Garmin uses the ANT+ protocol to communicate with external devices. ANT+ is a wireless technology that's designed for transmitting sensor data for physical activities or other health monitoring. You can mix and match any ANT+ devices, provided that they both support the same activities. If you've got a temperature monitor, and your wearable or handheld supports the temperature monitoring activity, the two can talk to each other.

NOTE

ANT is a low-power wireless protocol that defines how devices communicate with one another. ANT+ is a higher-level protocol built on top of ANT that defines a variety of device profiles (such as heart rate monitor). While ANT is a general wireless networking protocol, ANT+ defines the specific communication format (which channel configuration to use, how to structure data) for each specific type of device. This means that any ANT+ enabled device that is capable of talking to an ANT+ enabled heart rate monitor can, indeed, talk to any ANT+ enabled heart rate monitor regardless of the device manufacturer.

Custom Device Apps

This is where you get to interact directly with the users. A custom device app lets you display information, capture data, and get input directly from the user.

Your custom apps can offer up activities that the user can start, stop, and record. You can also interact with companion phone apps and web services. You'll use the Connect IQ SDK and developer tools to build these apps, which I discuss in Chapter 3.

Custom Tracking

The FIT (Flexible and Interoperable Data Transfer) format is a binary file format that tracks the values of fitness sensors along time and space. It includes your GPS tracks, the time at which each point on the track was sampled, and various data fields of interest along the way. For example, a biking activity would track heart rate, speed, and distance traveled at each point.

In addition to creating your own apps to record activities, and creating your own data fields to represent data that you collect, you can also record your own tracking data into the FIT format using FIT developer fields.

If you're planning on doing any integrations with the Garmin Connect API, you'll want to check out the FIT SDK.

Data APIs

In order to make things happen beyond the confines of your device, you're going to need to turn to APIs to make this possible. Garmin has its own APIs, the Garmin Connect API and the Garmin Wellness API (see Table 1-1). But through the power of OAuth, which allows apps and services to authorize access to one another, you can authenticate against a well-known authority such as Google, Facebook, or Twitter.

Before you can move data through an API, you need to know where it comes from. Here's how data flows through the system before it's able to reach the outside world: first, the device records the data. Now, you could just export the data as a FIT file, but where's the fun in that? The next thing that happens is that the data gets synced into Garmin Connect (often through the Garmin Connect app running on the user's phone). Once it's

in Garmin Connect, developers can connect the data and create experiences around it.

Table 1-1. Feature comparison between Connect API and Wellness API

Feature	Connect API	Wellness API
All Day Step Count		Y
All Day Calorie Count		Y
All Day Distance		Y
Sleep Duration		Y
All Day Heart Rate		Y
Index Scale Information		Y
Device ID	Y	Y
Start Time of Monitoring Period	Y	Y
Activity Type	Many	Many
Duration	Y	Y
Active Seconds	Y	Y
Steps	Y	Y
Distance	Y	Y
Calories	Y	Y
Intensity	Y	Y
METs	Y	Y
Heart Rate	Y	Y
Speed	Y	
Pace	Y	
Cadence	Y	
Power	Y	
GPS	Y	

Extended Community
: The full circle is not closed. In addition to extending out into APIs, you can also create experiences that help build communities around workouts. Users can share their data with others, take part in challenges, and reinforce other users' goals.

The path from a user's body into an API is well defined in the Garmin ecosystem. The full circle experience doesn't end with the API or with the extended community. The circle originates with the user,

but it also returns there in the form of notifications, visualizations, sharing, and application experiences.

ANT versus BLE?

ANT and Bluetooth Low Energy (BLE) may seem to be competing standards, but they each have their strengths in different applications. In ANT, each node has equal capabilities, whereas in BLE, the networks are asymmetric, with a hub-based approach (often with your phone or computer at the center).

Because ANT is a symmetric model, the requirements for a network are simplified. You can have multiple peers with relatively similar and low computing power requirements, while BLE requires a hub device with significant computational power.

BLE uses a star networking model with a hub/master device at the center, which coordinates each of the other devices on that network. ANT is able to accommodate that model, but also includes mesh networking. This means that ANT is able to scale to allow more sensors to be used at once, but it also allows sensors to talk to one another directly, without needing to communicate with the hub first.

While BLE theoretically has no limit on the number of devices that can participate in a network, there are implementation-specific limitations. For example, Android 4.4 is limited to 7 simultaneous connections, and 5.0 allows you to go up to 15.

While there are many overlapping use cases between BLE and ANT, ANT is particularly suited to fitness and health tracking. So if you wanted to create a solution where a single device was aggregating heart rate sensor data from multiple people (a baseball team or the crew of your starship), ANT would be the natural choice.

From the Body to the Cloud

The Garmin Connect API allows you to create systems that take in activity data from end users, and build on that data. For example, you could create a system that lets sports teams track the performance of the team as a whole by analyzing sensor data aggregated from individual players. You can also use the Garmin Connect API

to integrate with analysis tools, social media, or games. Examples of this are Strava and TrainingPeaks.

It's a cloud system for workout data. Access to an individual user's feeds are managed by OAuth, so users are in control of what data is available to a system you create, and they can revoke that access if they want to.

Garmin Health/Wellness

More and more, employers are correlating employee health to work–life balance, happiness, decreased healthcare costs, and improvements in productivity. Garmin provides a wellness program based on activity tracking from its fitness devices. This program allows employers to set specific fitness goals for their employees, measure them, and reward employees who reach their goals.

Through the Garmin Wellness API, which is available to approved developers, you can roll your own wellness program and take in data from Garmin devices. The API gives you full access to data (though it does require opt-in from employees) and the ability to build wellness solutions for in-house use, or as part of a product offering you create.

In addition to the API, there are Garmin partners you can work with (for example, Validic) who provide integration between activity trackers and your application, wearable device, or in-house systems.

With the Wellness API, Garmin provides access to fitness and activity data in the form of data exports that you can bring into your wellness solutions. Garmin's wellness solutions also offer hardware that makes it easier for employees to share their data. The vívohub can download data from users who have opted in when they walk by the device, avoiding the need for them to pair their device with an app to upload data to Garmin. To get started, check out Garmin's developer programs, where you will find all you need to get on your way.

In Chapter 2, we'll talk about the tools, design principles, and key platform features you will use as you create your own solutions in the Connect IQ ecosystem.

CHAPTER 2
The Connect IQ Platform

Connect IQ is a platform for third-party apps that run on Garmin devices. If you're developing a system that runs on the wearable device, either working with existing data fields or talking to an entirely new accessory that you are developing, you'll start with Connect IQ. There are several different things you can create with Connect IQ:

Watch faces
: These aren't just a pretty picture to look at, but they can be that, too. You can create custom watch faces for your company or organization. But you can also incorporate data into a watch face, and display progress toward specific goals the user has set.

Data fields
: As described in Chapter 1, these are data points that can be incorporated elsewhere into the Connect IQ platform full circle. A data field can come from a sensor accessory that you manufacture, or could be derived from other data fields.

Widgets
: These are mini apps that appear as users scroll through the carousel of widgets installed on their device. Widgets provide at-a-glance information but can also communicate with phone apps and web services, and also can pull in activity, location, and sensor data.

Device apps

: These are a full-blown interactive experience for the user. They can be apps that start and stop activity tracking, or interactive apps for collecting, viewing, or manipulating information the user is interested in.

Garmin follows what it describes as a "purpose-built device strategy" where the company creates devices specifically designed to enable active lifestyles of people around the world. This includes a wide range of wearables, bike computers, and handhelds. For a complete list of the compatible devices, go to the Garmin Developer page. For this book, I will focus on Garmin wearables.

Designing for Different Watches

Garmin watches come in different shapes and sizes, and are built to different purposes.

For example, there are round watches such as the fēnix 3, and semi-round watches (circle with flat top and bottom) like the Forerunner 230. There are square watches like the vívoactive and Forerunner 920XT. You'll be able to define a different layout for each type of watch.

There are devices built for runners: the Forerunner 235 is great for beginners, while the 735XT is built for elite runners. There are luxury watches like the fēnix Chronos, marine watches like the quatix, and watches for aviators like the D2. And the vívoactive HR is made for an active person who might not be preparing for a big race.

What this means for you, as the developer, is that each device has different capabilities and different constraints. For example, the vívoactive HR has a built-in, wrist-based heart rate sensor, but the quatix 3 does not. This doesn't mean you can't run an app that requires a heart rate sensor on the quatix 3, but the user would have to buy a heart rate sensor with ANT+ technology to be able to use your app.

On the flip side, the quatix 3 has a barometric altimeter for precise measurement, while the vívoactive HR can obtain altitude only through its GPS. So while both can provide the same data, they obtain them in a different way and have varying degrees of accuracy.

Understanding the variety of devices and their constraints will help you rise up to one of the big challenges in wearable design: how to create an app that works great on a lot of different devices. As you'll see in Chapter 3, the Connect IQ SDK provides abstractions for the platform's features, making things easier for you as a developer. For example, you don't need to know whether a heart rate sensor is built into the device or connected wirelessly via ANT+ technology.

Connect IQ Tools and Features

Connect IQ allows you to build several kinds of app types, which I introduced at the start of this chapter: watch faces, data fields, widgets, and device apps.

There are a lot of features you can take advantage of in these apps:

Graphics
: You can draw images on screen, include bitmaps, use fonts, and also embed graphs with data based on data fields.

Sensors
: You can access any sensor built into your watch, connect to one of the many ANT+ sensors, or even use a generic radio channel for simple communication scenarios.

Data recording
: You can capture, record, display, and share data in the FIT format.

Connectivity
: You'll be able to communicate with a mobile phone, which can act as your gateway to the Internet. Through this gateway, you can also connect to authenticated web services using OAuth to securely identify which services have access to your data and what they can do with it.

Phone Not Required

Although a phone is really useful for Connect IQ, it's not necessary. There are still ways to upload device data using a USB connection to your computer. But a smartphone makes a great complement to devices that run the Connect IQ platform. The Garmin Connect mobile app is available on iOS, Android, and Windows phones.

Use device APIs
: Connect IQ includes APIs for the user interface, calendar, GPS and other sensors, connectivity to mobile phones and wireless sensors, and local storage for storing information when your app isn't actively running.

App Store
: When you're ready to publish your app for the world to use, you can publish it to the Connect IQ App Store. The approval process generally goes quickly. You can browse the app store and see what other developers just like you have innovated with the Connect IQ platform.

  You can also download an app directly to your own device without needing to go through the app store. This is called *side loading*, and you'll learn how to do it in "Run an App on a Real Device" on page 20.

Designing for Wearables

A wearable device is not a phone. Its screen is a small fraction of the size of a phone; its CPU and battery won't stand up to heavy-duty computation. As a result, you need to change the way you think about app development. Here are some best practices for wearable app design:

You've got seconds of interaction
: Don't take a minute to explain what could be conveyed in a second. A bicyclist cannot safely ride while fiddling with a device; your app should consume no more time than looking in the rearview mirror.

Technology must be invisible
: A smartphone interrupts and demands your attention. A wearable, when it does its job right, augments its wearer. It must feel like a seamless extension of the self.

Glance-ability
: With only seconds of interaction, your app needs to be glanceable, just like a rearview mirror. If a bicyclist needs to know the temperature, time, or her heart rate, that information needs to be communicated in a way that can be absorbed instantly.

Contextual
: A truly smart app will surface the information that the user needs at a given moment, and only that information. Sure you've got buttons, and maybe a touchscreen. But don't make the user hunt for information. Use what you know about the user to make smart decisions about what to show. Is the user hurtling down a road at 20 miles per hour? Maybe you should bump up the font size.

Limited navigation and interaction
: Even with a touchscreen, even with buttons, you should be sparing in how much navigation and interaction you demand of your users. It makes sense to swipe to switch view modes, or to tap to start and stop an activity recording. But you shouldn't make your users crawl through menu after menu to make something happen.

There are two common varieties of use cases: designing for an activity and designing for all-day needs. A gym workout app, where the app is running while the user is involved in a strenuous activity, can't demand too much interactivity. You might tap the screen at the start of the workout. Once your workout begins, your app should record it, but also display important metrics, perhaps heart rate and elapsed time, during the workout.

On the other hand, an all-day app like a weather forecast is going to have different behaviors. That kind of app has the freedom to be a little more complex. It will connect to the network, it may present a top-level user interface with some basic information, but require user interaction to drill down. How you apply these practices will vary depending on your use case.

Though Connect IQ is full of powerful tools for the hacker in all of us to tinker with, there are opportunities for real business value by tying into the Garmin ecosystem and leveraging the Connect IQ platform:

Exposure
: With millions of devices around the world, you have the potential to get your app in front of users in a big way.

Engagement
: Connect IQ puts powerful tools into your hands so you don't have to create your own toolkit. Maybe it's a powerful, durable

screen to display your data, or a source for sensor data for your solution. There are lots of ways you can extend and enhance your solution.

Drive hardware sales
: Sensors and accessory devices are a key component to the power of the Garmin ecosystem. Through Connect IQ, you have many tools to connect your device to Garmin and raise awareness of your device.

Generate revenue
: The Connect IQ platform and its tools are built for guiding your users through your existing business models. Premium memberships, in-app purchases, and other models can all be activated through connectivity. Tools like the Popupwebpage API, where you can initiate a message on the phone to open a web browser, let you guide the user to "learn more," set up an account, and otherwise engage with your services and brand.

In Chapter 3, you'll learn how to get started developing simple apps with Connect IQ. In Chapter 4, you'll see how you can create apps that go beyond the confines of the wearable device, and interact with sensors and the cloud.

CHAPTER 3

Getting Started with Connect IQ

It's time to learn how to build an app. The Connect IQ SDK lets you write code for Garmin wearables with a minimum of fuss and few lines of code. With starting templates for four kinds of apps, plenty of sample code, and a programming language (Monkey C) that is similar to Java, JavaScript, PHP, Ruby, and Python, you'll be up and running quickly. You'll need to configure some software first, and (optionally) get your hands on some hardware. You can download the free SDK from the Garmin website.

What You'll Need

Garmin wearable
: This is optional. The Connect IQ SDK includes a simulator that can run any code you develop. However, there's no substitute for testing on a real device. You can find a list of compatible devices at the Garmin Developer site.

ANT+ connectivity
: In Chapter 4, you'll learn how to go beyond this chapter's simple example. One of the projects shows how a Garmin wearable can connect to an external sensor using the ANT+ wireless protocol.

  If you don't have a compatible device with Connect IQ that supports the Temperature profile, you'll need to run the app in the simulator with a Garmin ANT+ USB adapter (Garmin part

number 010-01058-00, available from a variety of resellers including Garmin, Walmart, and Amazon).

The temperature sensor you'll use is the Garmin tempe sensor (part number 010-11092-30).

Computer
: The Connect IQ SDK will run on a Windows or Mac computer.

  You'll need one USB 2.0 or 3.0 port. If you are running code on a Garmin wearable, you'll use USB to transfer your programs to the device. If you are running the ANT+ project in the simulator, you'll need to connect the ANT+ USB adapter.

Install the SDK

To get up and running, check out the Getting Started guide, which includes links for the Connect IQ SDK. Follow those instructions, installing the SDK and the Eclipse IDE as directed in that guide.

Here are a few things to know about working with Eclipse:

Workspaces
: The first time you run it, Eclipse will ask you to choose a workspace location (where all your project should be stored). It will default to a directory named *workspace* in your home directory. If you don't expect to be partitioning your projects into multiple workspaces, you can select the box labeled "Use this as a default and do not ask again." Until you check that box, Eclipse will ask you to choose your workspace each time you start it.

Plug-in security warnings
: When you install the plug-in, you may receive a warning that you're installing software that contains content that hasn't been signed with a security certificate. You'll need to allow this in order to complete installing the plug-in.

What to do when your window layout gets messed up
: At one point in the installation instructions, you'll be told to enable the Connect IQ perspective, which includes many useful settings and windows, such as the Project Explorer. If you close the Project Explorer, there are a couple of ways to get it back. One is to choose Window→Show View→Other, which brings up the Show View dialog. Choose Project Explorer under the Other option and click OK. You can also choose Window→Per-

spective→Reset Perspective, which gives you the option to reset the Connect IQ perspective to its defaults.

Create a New Connect IQ Project

With Eclipse and the Garmin Connect IQ SDK installed and configured, you're ready to create your first project. This is where you'll start when you want to create a new project.

1. In Eclipse, select File→New→Connect IQ Project. A window will appear asking you to choose a name. Give your project a name (for this tutorial, name it "MyFirstApp"), then click Next.
2. Next, you'll be prompted to choose the project type, minimum SDK version, and target platforms. Choose Watch App, version 1.2.x, Square Watch, and Tall Watch, as shown in Figure 3-1. If you own a Connect IQ device, choose it here as well. That way, you'll be able to run the app on your watch later (the Square Watch and Round Watch targets will only work in the simulator).

   Click Next (don't click Finish yet).

> **NOTE**
>
> There are several other project types available to you, including Watch Face, Data Field, and Widget. For an overview of each option, see the Garmin Developer site.

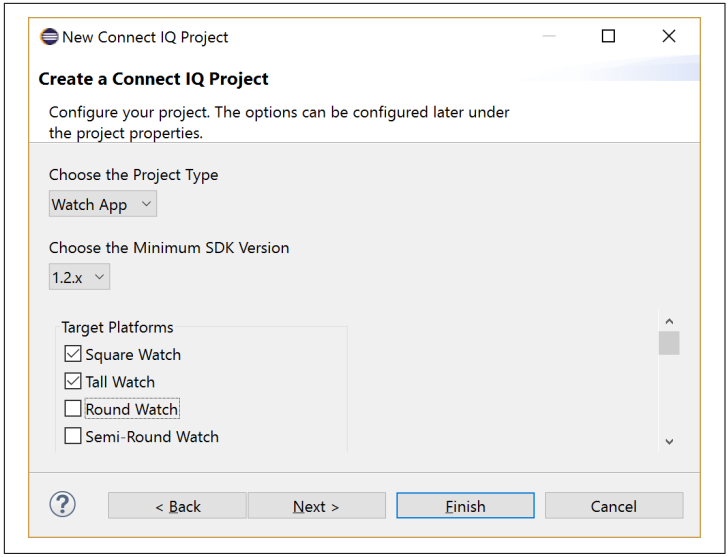

Figure 3-1. Choose these options for your first project

3. The next screen that appears doesn't give you any options aside from Simple (the type of the app). Click Next.
4. This page lets you specify which languages to support. Choose the language(s) you intend to support, then click Finish.

> **NOTE**
>
> If you don't see your project appear, close the Eclipse Welcome tab.

5. Now you need to set up a Run Configuration for your project, which configures how it appears when you test it. Click Run→Run Configurations, select Connect IQ App, then click the New Launch Configuration toolbar button just above the list of options.
6. Give your configuration a name (such as "MyFirstApp_Configuration"). Select your project from the list of projects and choose Square Watch for the target device type. Next, click Apply, then click Close.

As you create more apps, you'll create new Run Configurations for them, so this list will get bigger over time.

Run an App in the Simulator

Now you're ready to run this bare-bones app in the simulator. Each time you want to run an app in the simulator, you need to follow these steps:

1. Click the Connect IQ menu, then choose Start Simulator, and wait for the simulator to start.

   NOTE

   On Windows, you may get a prompt from Windows Firewall asking what kind of network access to give to the simulator. Choosing access on Private Networks should be fine, unless you plan to develop and test your app while connected to an untrusted network (such as at a coffee shop), in which case you should also select Public Networks. Click Allow Access.

   On macOS, you may get a warning that the ConnectIQ app was prevented from running because it is from an unknown developer. If you see this, go to the Finder, navigate to the *bin* subdirectory of the Connect IQ SDK, right-click on the ConnectIQ app, and click Open. macOS will still warn you, but it will let you run it. The next time you try to run the simulator, you won't get the error message.

2. Finally, return to Eclipse, click the Run menu, and select Run Configurations. Select the configuration you created earlier, and click Run, as shown in Figure 3-2.

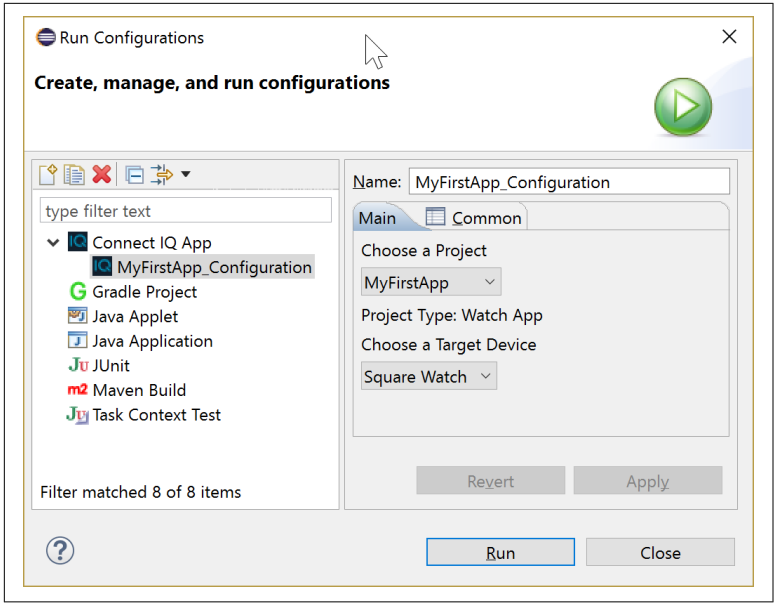

Figure 3-2. Running your first app

Run an App on a Real Device

You can also *side load* your app onto a real device. To do this:

1. Plug your device into your computer over USB.
2. Click Connect IQ→Build Project for Device. In the wizard dialog that appears, choose the name of the project, which device to build for, and where to put the compiled program (it must be in the *GARMIN/APPS* folder of your device). The signing key will default to the signing key you created when you configured the SDK (see "Install the SDK" on page 16).
3. The wizard should look like Figure 3-3. Click Finish. The wizard won't close by itself, so you'll need to close it.

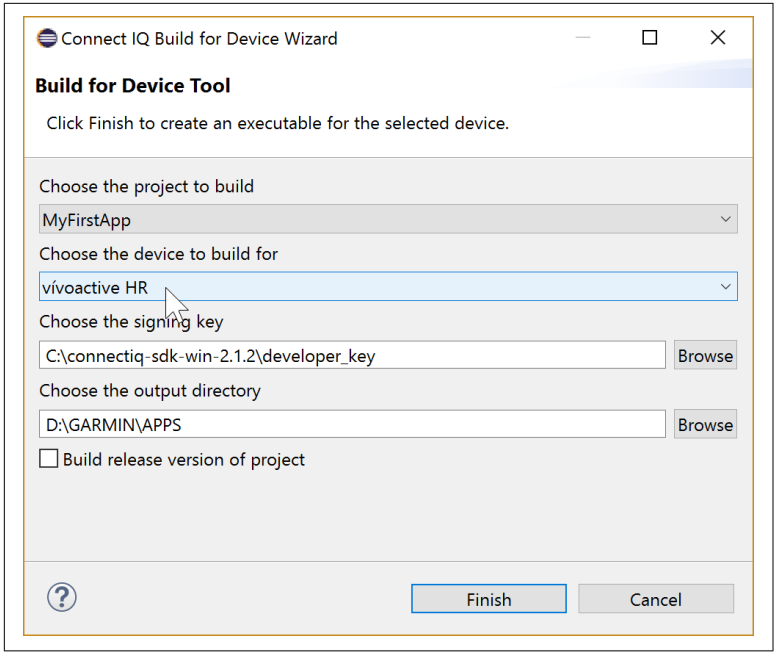

Figure 3-3. Building for a real device

NOTE

If you don't see your device type listed, right-click on your project in Project Explorer, choose Properties, and select Connect IQ. This will open up the Project Details options you saw earlier in "Create a New Connect IQ Project" on page 17 where you can select additional target platforms.

Disconnect your Garmin device from your computer (if you're on a Mac, you should eject the drive before you unplug it), and go to the list of programs on your device. Find your app (in this case, MyFirstApp), and run it.

Exploring the App

Let's take a look at this sample app. If you ran it in the simulator, it will look and work great. Figure 3-4 shows how it looks in the square watch simulator. You can click the menu button (third icon from the left in the image) and it will pop up a two-item menu.

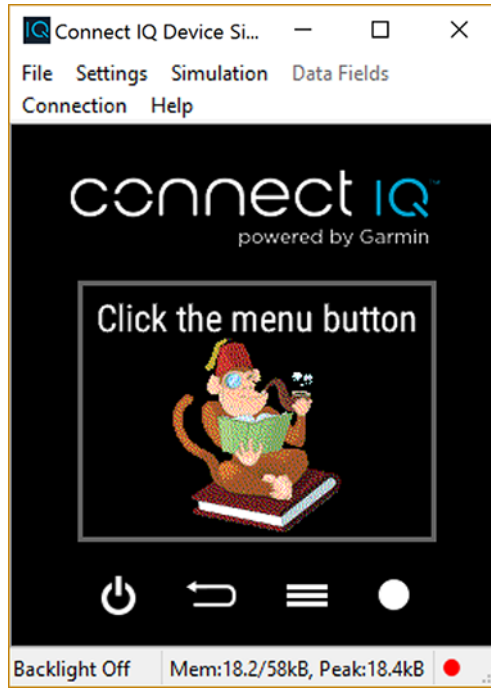

Figure 3-4. Running the sample watch app in the simulator

But, if you were to run this on a watch with a smaller screen, you might have a problem. Figure 3-5 shows how the app looks in the simulator when it's configured to behave like the vívoactive HR GPS smartwatch.

Figure 3-5. The sample app on the vívoactive HR

Configure the App for a New Device

To run the app in the simulator as shown in Figure 3-5, you need to configure it for the vívoactive HR device and also add a new Run Configuration:

1. In Project Explorer, right-click on MyFirstApp and choose Properties. Choose Connect IQ from the list on the left, and then check the box for vívoactive HR under Target Platforms. Click OK.
2. Next, click Run→Run Configurations. Make sure Connect IQ App is selected in the list of configurations, then click the New Launch Configuration icon above the list of configurations.

3. Name the configuration "MyFirstApp_vivoactiveHR", select MyFirstApp under Choose a Project, then choose vívoactive HR under Target Device. Click Apply, then click Run.

From here on out, you can run it under this configuration by choosing Run→Run Configurations, selecting MyFirstApp_vivoactiveHR, and clicking Run. It should look just like Figure 3-5.

NOTE

You can leave the simulator running when you're done checking it out. The next time you run an app in the simulator, it will stop whatever app you're running and start the new one. This means you won't have to wait for the simulator to launch each time.

How the Source Code Is Organized

Before I show you how to modify the app so it looks great on the vívoactive HR, I want to show you around all the files in the app.

First, locate MyFirstApp in Project Explorer. Expand the folder view so you can see all the folders and all the content in them. If you're on Windows with a numeric keypad, you can press the * key on the numeric keypad to expand all the folders. On Mac, Alt-Right Arrow will do the same thing. Figure 3-6 shows Project Explorer with everything expanded, providing an overview of what you'll see in the MyFirstApp project.

Your first question may well be, where did all this stuff come from? When you selected the Watch App project type, this instructed Eclipse to copy in all these files from a template. So, every time you create a new app and choose Watch App, you'll find all the same files here.

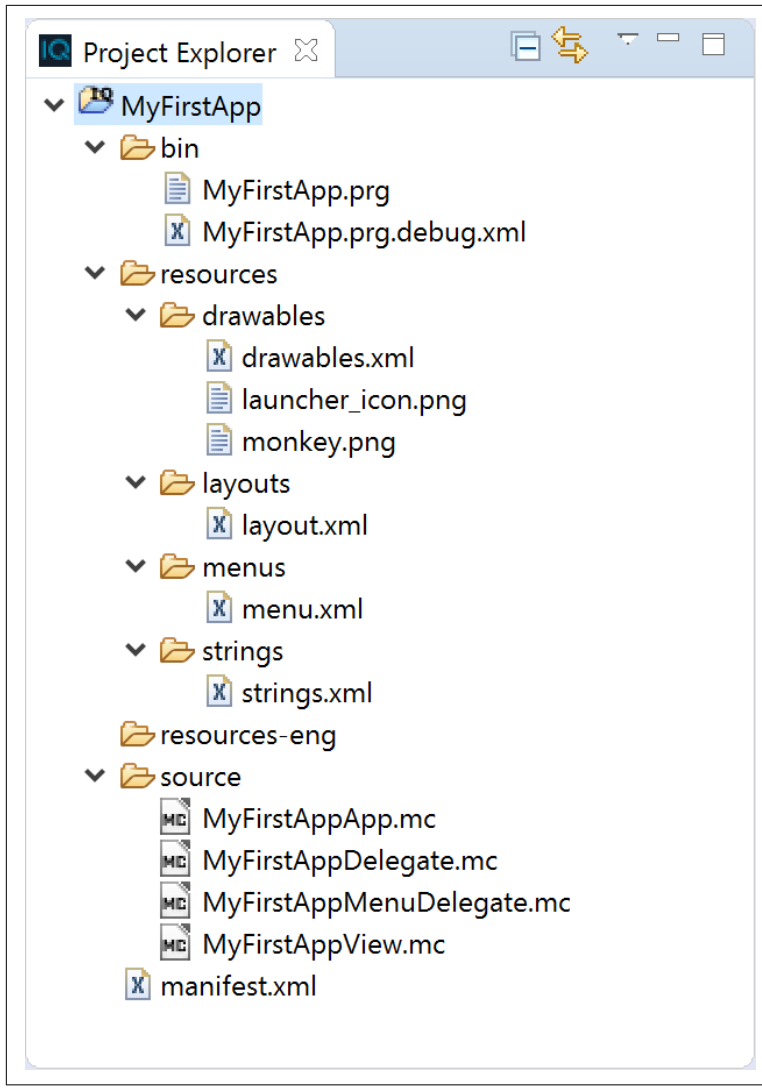

Figure 3-6. All the files from the Watch App template

There are four top-level folders under you app, and one file:

bin
: This is where all the compiled versions of your app go. Compiled apps have the extension *prg*, and this is what gets copied to your device when you run it on a real device.

resources
: This is a collection of XML files that define the appearance of elements on the screen, the text of buttons and labels, and other user interface elements.

resources-eng
: This is a *localized* resources folder. If you wanted to override any elements in *resources* for a specific language, you'd put customized versions of the resource XML files here. Because you've configured this app for only one language (English), and because all the English-language resources are in the *resources* folder, you don't need to do anything with this folder.

source
: This is where all the Monkey C sources files reside.

manifest.xml
: This file contains all the configuration details specific to your app.

Fix the Font Size

Your first task, fixing the font, will take you to the *resources* folder. You could simply edit the *layout.xml* file and use a smaller font than the default, but that would look too small on a square watch. Instead, you'll create a separate layout that only gets used when on a device with the same screen size as the vívoactive HR (148×205 pixels).

NOTE You can download this and other sample code from Garmin's GitHub repo.

1. Right-click on MyFirstApp, and click New→Folder. In the window that pops up, make sure that the parent folder is *MyFirstApp*. Give it the name *resources-rectangle-148x205* and click Finish. The resources folders are also used for specific device layouts and will be discussed later in more depth.

> **NOTE**
>
> It is essential that you name it exactly as shown. In order to override a resource, you need to use the name *resources*, add a hyphen, and follow it with one of the supported qualifiers—in this case, *resources-rectangle-148x205*. The Resource Compiler Guide has more details.

2. Under the original *resources* folder, right-click *layouts*, then choose Copy. Then, right-click the *resources-rectangle-148x205* folder, and choose Paste.
3. Expand the *resources-rectangle-148x205* folder and its child *layouts* folder. Double-click *layout.xml* to open it in the editor window.
4. In the editor window, expand the node named label, then right-click it and select Add Attribute→New Attribute. Give it the name `font`, and the value `Gfx.FONT_TINY` and click OK.
5. Expand the node named bitmap, then edit the filename field and change it from ../drawables/monkey.png to ../../resources/drawables/monkey.png. This will let you use the *monkey.png* image from the *resources* folder without needing to copy it into *resources-rectangle-148x205*.
6. Choose File→Save to save the *layout.xml* file.

Your project should now look like Figure 3-7.

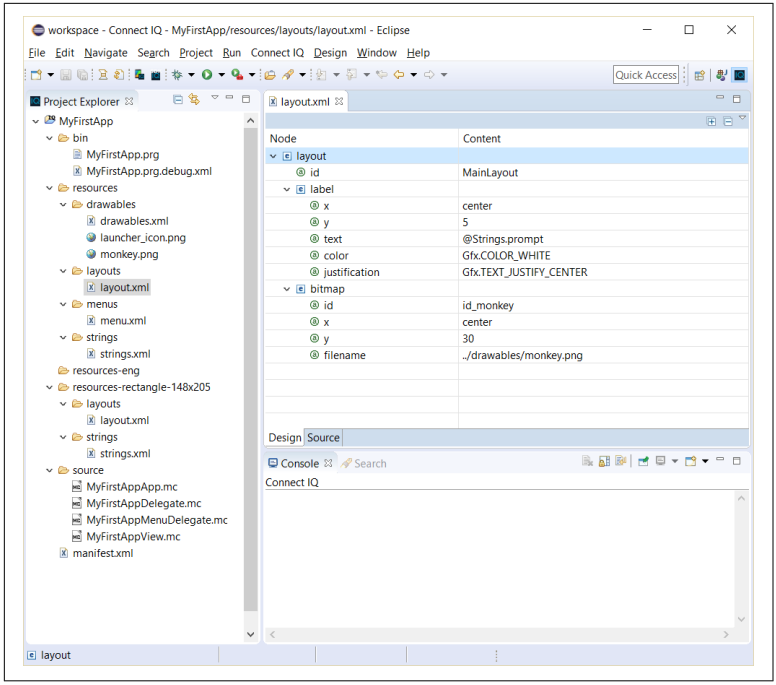

Figure 3-7. The project with the rectangular 148x205 layout added

Now it's time to see if it worked. First, click Run→Run Configurations, choose the MyFirstApp_Configuration, which uses the Square Watch as its target device, and click Run. The font should still be nice and big. Next, run it with the MyFirstApp_vivoactiveHR configuration, and the text should now fit on the screen as shown in Figure 3-8.

Figure 3-8. The app with smaller text on the vívoactive HR

Menu Button?

This is a lot better. However, there's now the matter of the prompt "Click the menu button." The vívoactive HR has no menu button. Instead of using a menu button, you long-press on the button on the right. Let's change that prompt, "Click the menu button." To change it, you need to add another resource to the *resources-rectangle-148x205* folder:

1. Right-click the *resources-rectangle-148x205* folder, and choose New→Folder. Name this folder *strings* and click Finish.

2. Right-click the newly added *strings* folder and choose New→File. Name the new file *strings.xml* and click Finish. The file will open in the editor.
3. Right-click in the editor window, and choose Add Child→New Element. Name that element "strings" and click OK.
4. Right-click on the new "strings" element, and choose Add Child→New Element. Name that element "string" and click OK.
5. Right-click on the new "string" element, choose Add Attribute→New Attribute, and name it id. Set its value to prompt. Right-click the "string" element again, choose Add Child→#PCDATA;, then change the content of the newly added child from "strings" to "Press/hold right button".
6. Click File→Save to save *strings.xml*.

Next, run the project with the MyFirstApp_vivoactiveHR configuration, and you should see the new prompt appear.

A Tour of the Source Code

Let's have a look at the code that makes this run.

In the *source* folder, there are four files that contain this app's code. All of the source code files contain a class, which contains several methods. One of these, `initialize()`, calls the `initialize()` method of its base class, and is present in all four files/classes.

MyFirstAppApp.mc
: This is the *entry point* of your application. There are three additional methods in here aside from `initalize()`: `onStart()`, which gets called when the app first begins; `onStop()`, which is called when the app runs; and `getInitialView()`, which sets up the first view/screen you see when the app starts (in this case, a Monkey with the text "Click the menu button").

MyFirstAppDelegate.mc
: This is the app's *delegate*, which contains all the methods that define the behavior of the app. This contains the `onMenu()` method, which displays a menu when the device's menu button is pressed, or when another action takes place (such as long-pressing the right button) that maps to the menu button action.

The menu that's displayed is defined in the *resources/menus/menu.xml* file and contains two menu items.

MyFirstAppMenuDelegate.mc
: This is the delegate for the menu, and contains a menu handler method, `onMenuItem()`, which gets called when the user selects something from the menu.

MyFirstAppView.mc
: This is responsible for controlling the layout of the initial view and is referred to in *MyFirstAppApp.mc*'s `getInitialView()` method. The `onLayout()` method retrieves the `MainLayout` through the `Rez` class, which contains objects taken from the resource XML files. Remember how you modified the *layout.xml* file earlier? If you look at that file (both of them, in fact), you'll see that the layout has the ID `MainLayout`, which corresponds to what you see in the `onLayout()` method. This `MainLayout` object has different properties at runtime, depending on which watch type you're using, based on the changes you made in the previous section.

Now that you know how to create and customize Connect IQ apps, you're ready to tackle some projects.

CHAPTER 4

Projects

There's no better way to learn something new than by example. I've put together some simple projects that you can follow along with to learn some skills for Connect IQ. First, you'll learn how to run the example programs that come with the Connect IQ SDK, and also how to modify one of those examples. Next, you'll step out of the physical confines of the simulator and use ANT+ technology to talk to an external sensor, the Garmin tempe temperature sensor. Finally, you'll learn how to communicate with an authenticated web service.

NOTE You can download these projects and other sample code from Garmin's connectiq-apps GitHub repo.

Personal Data Tracking

The Project Explorer in the Connect IQ perspective shows all the Connect IQ projects in your workspace. Unless you've been exploring on your own, you should only see one project, MyFirstApp. Keeping track of which project an open file corresponds to can be tricky, so I suggest you right-click on MyFirstApp and choose Close Project. That will collapse its folders and also will close any of its files you may have open. Do the same with any other projects you're working on.

1. Now you're ready to import one of the Connect IQ sample projects. Choose File→Import, and the Import dialog will appear. Make sure General→Existing Projects into Workspace is selected as shown in Figure 4-1. Click Next.

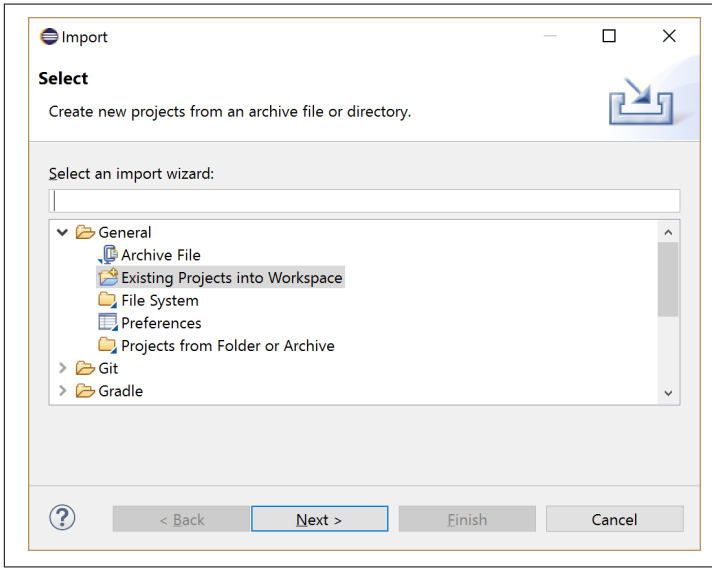

Figure 4-1. Importing an existing project

2. In the next page of the dialog, click Browse next to Select Root Directory, and make your way into the folder where you extracted the Connect IQ SDK. In the *samples* subdirectory, choose RecordSample (don't choose any of its subdirectories) and click OK.
3. Back in the Import dialog, check the box to the left of Copy Projects into Workspace. That way, you'll be working on a copy of the sample (very important in case you make any mistakes) rather than the original. The Import dialog should look like Figure 4-2. Click Finish. The RecordSample project will appear in your Project Explorer.

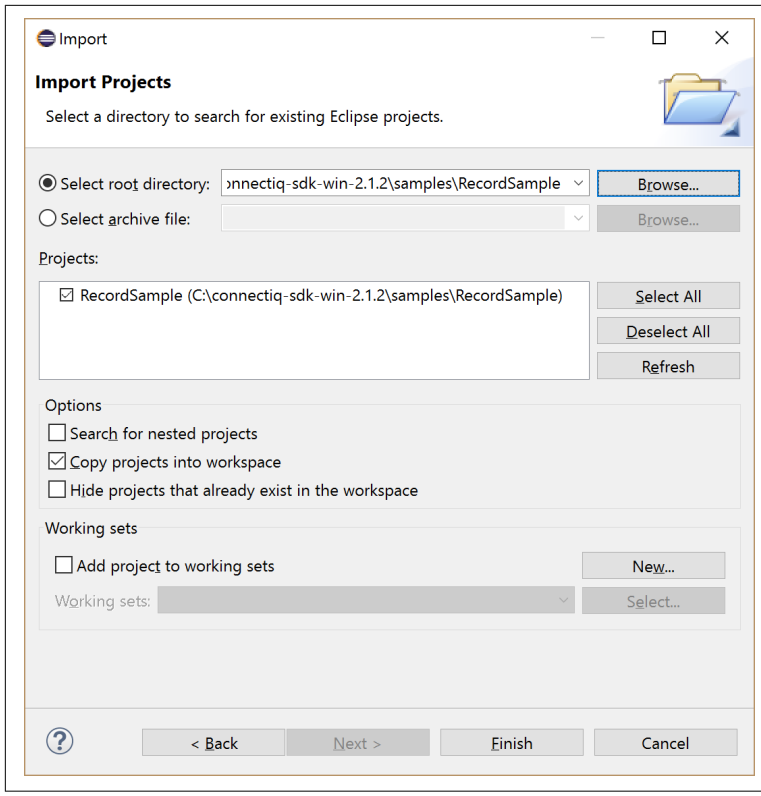

Figure 4-2. Importing the RecordSample

This project is a simple track recorder. If you run it on your watch or your simulator, it will generate a FIT file with GPS locations. It's preconfigured for the Square Watch format as well as a few other devices. To run it, right-click on the project, then choose Run As→Connect IQ App.

NOTE

The Run As option lets you bypass creating a Run Configuration. But you'll be asked to choose what kind of device to run it on when you launch it as shown in Figure 4-3, so you may also want to create a Run Configuration for convenience (see "Create a New Connect IQ Project" on page 17).

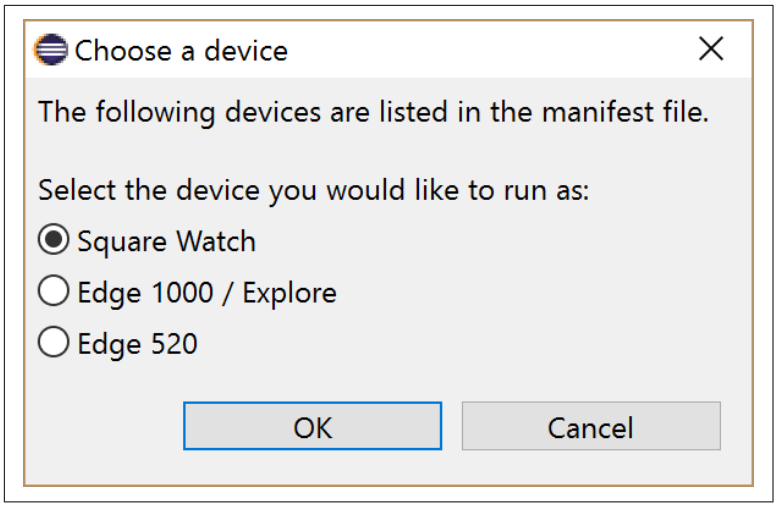

Figure 4-3. Choosing what kind of device to run as

Choose the Square Watch option, and when the simulator starts, you'll see a blank screen with the text "Press Menu to Start Recording." Press the menu button on the simulator, and it will start recording data. You can press the menu button again to stop the recording. Let it run for a minute or two and then use the button again to stop.

NOTE

On devices without a menu button, such as the vívoactive HR, long-press on the right button to start and stop the recording.

You can also run the sample directly in your device with the Connect IQ→Build for Device Wizard option in Eclipse.

NOTE

If your device is not one of the few devices supported by the sample, right-click RecordSample in Project Explorer, choose Properties, make sure Connect IQ is selected in the lefthand side of the screen, check the box for your device, then click OK.

Access the FIT File

When you're done recording, select Simulation→FIT Data→Save Fit Session to save the *FIT* file to a location on your computer.

If you ran the sample on your device, you can later retrieve the *FIT* file from your device. Connect it to your computer with the USB cable, and look in the *GARMIN/ACTIVITY* folder.

The FIT SDK includes some programs you can run to convert FIT files to other formats, such as comma-separated values. But if you've paired your device with the Garmin Connect App, and if you're signed up for an account, you can view your recording by clicking the menu icon and choosing Activities, as shown in Figure 4-4 (be sure to sync your device from your phone first). You'll then see a list of activities—in this case, "*CityName* Walking." Click on an activity to see all the information that the sample app recorded.

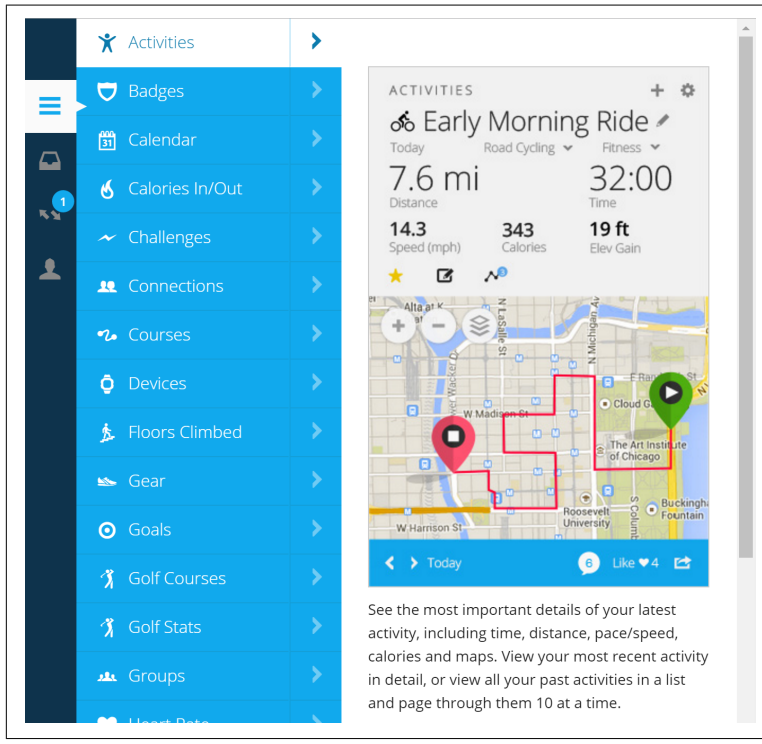

Figure 4-4. Selecting your activities on Connect

Add a New Field

What if you want to track additional data into the FIT file? Here are some quick changes you can make to the sample app to record a numeric value. First, open up *RecordSampleView.mc* in Project Explorer, and make the following changes.

Right below `using Toybox.ActivityRecording as Record;`, add these two lines so you can use classes and objects from these packages:

```
using Toybox.FitContributor as Fit;
using Toybox.Timer as Timer;
```

Next, you'll need to add the following lines right under the line `class BaseInputDelegate extends Ui.BehaviorDelegate`:

```
var mFooField;
var mFooValue = 0;
var fooTimer = null;

function updateFoo() {
    mFooValue = mFooValue + 1;
    mFooField.setData(mFooValue);
}
```

The first three lines add three new variables to the `BaseInputDelegate` class: one that represents the new data field, another for its underlying value, and the third for a `Timer` object that will be fired periodically. The `updateFoo()` method is a new method that the `Timer` object will call for you. It just increments the value by 1 each time.

Next, in the `onMenu()` function, add the following lines right before `session.start();` (this creates a new field that's recorded into the FIT file, sets its value to the value of `mFooValue`, and then sets up and starts a timer to periodically update the value and the field):

```
mFooField = session.createField("Foo", 0, Fit.DATA_TYPE_FLOAT,
          { :mesgType=>Fit.MESG_TYPE_RECORD, :units=>"bars" });
mFooField.setData(mFooValue);
fooTimer = new Timer.Timer();
fooTimer.start(method(:updateFoo), 1000, true);
```

Just below those lines, right before `session.stop()`, add the following line:

```
fooTimer.stop();
```

That line ensures that the timer is gracefully terminated before the recording stops.

Before you try to run this app, right-click RecordSample in Project Explorer, select Properties, go to Connect IQ, and check FitContributor under Permissions. Click OK.

Permissions

Because users and devices have such an intimate connection, the device knows a lot about an individual user. Most of this information is sensitive, so Connect IQ will not let your app access this information without permission. When you develop your app, you need to specify which permissions it needs.

When users install your app, they'll be notified of which permissions your app has requested. If they are not comfortable granting those permissions, they may decide to not install your app, or may not use certain features in it. So you should only specify the permissions your app actually needs here.

Here's the part of the *RecordSampleView.mc* file that you modified, with new lines shown in bold (some long lines have been wrapped from the original so it fits on the page):

```
using Toybox.WatchUi as Ui;
using Toybox.Graphics as Gfx;
using Toybox.System as Sys;
using Toybox.Lang as Lang;
using Toybox.ActivityRecording as Record;
using Toybox.FitContributor as Fit;
using Toybox.Timer as Timer;

var session = null;

class BaseInputDelegate extends Ui.BehaviorDelegate
{
  var mFooField;
  var mFooValue = 0;
  var fooTimer = null;

  function updateFoo() {
    mFooValue = mFooValue + 1;
    mFooField.setData(mFooValue);
  }
  function onMenu() {
    if( Toybox has :ActivityRecording ) {
```

```
        if( ( session == null )
            || ( session.isRecording() == false ) )
        {
          session = Record.createSession({:name=>"Walk",
                          :sport=>Record.SPORT_WALKING});
          mFooField = session.createField("Foo", 0,
            Fit.DATA_TYPE_FLOAT,
            { :mesgType=>Fit.MESG_TYPE_RECORD, :units=>"bars" });
          mFooField.setData(mFooValue);
          fooTimer = new Timer.Timer();
          fooTimer.start(method(:updateFoo), 1000, true);
          session.start();
          Ui.requestUpdate();
        }
        else if( ( session != null ) && session.isRecording() ) {
          fooTimer.stop();
          session.stop();
          session.save();
          session = null;
          Ui.requestUpdate();
        }
      }
      return true;
    }
  }
```

If you look inside one of the generated FIT files, you'll see the Foo field being populated. Here's an excerpt from a FIT file that I recorded on a vívoactive HR with this app, and converted to CSV using the *java/FitToCSV.bat* utility in the FIT SDK:

Field 12	Value 12	Units 12
Foo	5	bars
Foo	6	bars
Foo	7	bars
Foo	8	bars
Foo	9	bars
Foo	10	bars
Foo	11	bars

Notice how it simply increments, which is consistent with what's in the `updateFoo()` method.

Next, you need to add some string labels to the *strings.xml* file and create a *fitcontributions.xml* file to instruct Connect IQ as to how to graph these values. Expand the *resources* folder under *RecordSample*

and double-click *strings.xml*. In Chapter 3, you used the Eclipse XML visual editing tools to modify resource files. This time, I'd like you to try editing the raw XML. Click the Source tab under the editor window, and you should see this in the editor:

```
<resources>
    <string id="AppName">RecordSample</string>
</resources>
```

Add three new lines, one for the field label, one for the graph label, and one for its units. Your *strings.xml* file should now look like this:

```
<resources>
    <string id="AppName">RecordSample</string>
    <string id="foo_label">Foos</string>
    <string id="foo_graph_label">Foos</string>
    <string id="foo_units">bars</string>
</resources>
```

Save the file. Next, right-click the *resources* folder, add a new folder called *contributions*, then add a new file to that folder called *fitcontributions.xml*. Use the Source tab of the XML editor to add the `fit Contributions` resources as shown:

```
<fitContributions>
    <fitField id="0" displayInChart="true" sortOrder = "0"
    precision="2" chartTitle="@Strings.foo_graph_label"
    dataLabel="@Strings.foo_label" unitLabel="@Strings.foo_units"
    fillColor="#FF0000" />
</fitContributions>
```

Save the file, and run the app on your device. You won't be able to see the new field appear on Garmin Connect, because your app needs to be approved and published in the Connect IQ store for custom fields to appear. However, if you run the monkeygraph utility (Connect IQ→Start Monkeygraph), you can see how the graph would look. Before you run monkeygraph, go back to Eclipse and choose Connect IQ→App Export Wizard. This will create an *IQ* file that could be submitted to the Connect IQ store. But it also contains important metadata that monkeygraph needs to be able to graph the values. Select the RecordSample project, and export the IQ file somewhere.

Next, run monkeygraph, then use File→Open IQ File to open the file you just created. After that, use File→Open FIT File to open one of the FIT files you recorded after running the app. You'll see a graph like the one in Figure 4-5.

Figure 4-5. Graph of your foos as measured in bars

Read Data from a Sensor

Now let's experiment with talking to a sensor using ANT+ technology, the Garmin tempe temperature sensor. This project won't be an activity recorder; it will display the values, but won't record them.

1. Choose File→New→Connect IQ Project. Name it TempeApp, make it a Watch App, and choose Square Watch as the target platform (along with any other platforms you want to use).
2. Delete the following files or directories from the project (right-click, then choose Delete): *resources/menus*, *resources/layouts*, and *source/TempeAppMenuDelegate.mc*. These files relate to portions of the sample app (specifically, the part that displays the sample menus) that you won't need for this project.
3. Open *source/TempeAppDelegate.mc* in the editor, and remove the entire `onMenu()` function. The file should look like this when you're done:

```
using Toybox.WatchUi as Ui;

class TempeAppDelegate extends Ui.BehaviorDelegate {

    function initialize() {
```

```
            BehaviorDelegate.initialize();
        }

    }
```

4. Right-click TempeApp in Project Explorer, select Properties, and check ANT under Permissions. Click OK.
5. Double-click the *source/TempeAppView.mc* file to open it in the editor. Replace its contents with the code shown in Example 4-1. Save the file.
6. Right-click the *source* folder, choose New→File, name it *TempeSensor.mc*, and put the code shown in Example 4-2 in the file. Save it.

Example 4-1. Source of the TempeAppView.mc file

```
using Toybox.WatchUi as Ui; ❶
using Toybox.Graphics as Gfx;
using Toybox.System as Sys;

class TempeAppView extends Ui.View { // The class for the app view

  ❷
  var mTemp = 0;          // current temperature
  var tempTimer = null;   // timer for update frequency
  var mSensor;            // sensor object

  function updateTemp() { ❸
    mTemp = mSensor.data.currentTemp; // Read the temperature
    Ui.requestUpdate();               // Request a screen update
  }

  function initialize() { ❹

    // Create and start a timer.
    tempTimer = new Timer.Timer();
    tempTimer.start(method(:updateTemp), 10000, true);

    try {
      // Create the sensor object and open it
      mSensor = new TempeSensor();
      mSensor.open();

    } catch( e instanceof Ant.UnableToAcquireChannelException ) {
      Sys.println( e.getErrorMessage() );
      mSensor = null;
```

```
    }
    View.initialize(); // Initialize the UI
  }

  // Load your resources here
  function onLayout(dc) { ❺
  }

  // Called when this View is brought to the foreground.
  function onShow() { ❻
    // (re)start the timer
    tempTimer.start(method(:updateTemp), 10000, true);
  }

  // Update the view
  function onUpdate(dc) { ❼
    dc.clear(); // Clear the display

    // Set a white color for drawing and draw a rectangle
    dc.setColor(Gfx.COLOR_WHITE, Gfx.COLOR_TRANSPARENT);
    dc.fillRectangle(0, 0, dc.getWidth(), dc.getHeight());

    // Draw black text against a white
    // background for the temperature
    dc.setColor(Gfx.COLOR_BLACK, Gfx.COLOR_WHITE);
    dc.drawText(dc.getWidth()/2, 0, Gfx.FONT_XTINY,
                "Temperature:"+ mTemp, Gfx.TEXT_JUSTIFY_CENTER);
  }

  // Called when this View is removed from the screen.
  function onHide() { ❽
    tempTimer.stop(); // Stop the timer
  }

}
```

❶ Here's where you declare which libraries you are using. The `as` keyword lets you assign a shorter alias for them. So instead of writing `Graphics`, I can write `Gfx` throughout this file.

❷ These three variables represent the current temperature read from the sensor, a timer object for updating the value periodically, and a sensor object to refer to the sensor itself.

❸ This function is called by the timer, and does two things: gets the latest value from the sensor and forces the user interface to refresh, which causes the `onUpdate()` function to be called.

❹ This function gets called when things start up. It creates a timer that fires every 10 seconds, and also attempts to open the sensor, which is represented by the `TempeSensor` class.

❺ The `onLayout()` method is called when your app starts, and sets up the layout of your app's view. Because you're only drawing directly on the screen in the `onUpdate()` function, as opposed to displaying UI elements whose contents are refreshed, you don't have any layout to perform. The `dc` argument is short for "drawing context," which is an object that's provided to your app for all drawing operations. You want to put something on the screen? You go through the `dc`.

❻ The `onShow()` method is called when the app comes back into the foreground. This restarts the timer (it will be stopped in `onHide()`). This reveals a key part of power management. The device runtime will notify your app when the user sends it to the background. It does this by calling the `onHide()` method. When the user returns to the app later, the device calls your `onShow()` method.

❼ This function clears the screen, puts a white background on there, and adds some text with the current temperature read from the sensor.

❽ This stops the timer when the app gets sent to the background. See the preceding discussion for the `onShow()` method.

Example 4-2. Source of the TempeSensor.mc file

```
using Toybox.Ant as Ant;
using Toybox.System as System;
using Toybox.Time as Time;

class TempeSensor extends Ant.GenericChannel
{
  ❶
  const DEVICE_TYPE = 25;  // The ANT+ device type
  const PERIOD = 65535;    // How often we expect new data

  hidden var chanAssign;   // The channel assigned by the radio

  var data;                // Stores the data received from the
                           // sensor
```

```
var searching;           // Whether we're currently searching for
                         // the sensor
var deviceCfg;           // Device configuration details

// This flag indicates we've obtained enough
// data to read the temperature
var tempDataAvailable = false;

class TempeData ❷
{
  var currentTemp;
  function initialize()
  {
    currentTemp = 0;
  }
}

class TempeDataPage ❸
{
  static const PAGE_NUMBER = 1;

  function parse(payload, data)
  {
    // The payload (what we received from the sensor) has
    // a few data points in it. We're just interested in
    // the current temperature.
    data.currentTemp = parseCurrentTemp(payload);
  }

  hidden function parseCurrentTemp(payload) ❹
  {
    // Mask most significant byte (MSB) to see if it's > 127
    var intHigh = payload[7] & 0x80;

    // Combine the most significant and least significant bytes
    var currentTemp = (payload[7] << 8) | (payload[6] & 0xff);

    // If the MSB is over 127, invert its bits and multiply by -1
    if (intHigh > 0) {
      currentTemp = (~currentTemp & 0xFFFF) * -1;
    }
    currentTemp = currentTemp / 100f; // Divide by 100 to get
                                      // actual temp
    if ((currentTemp < -327) || (currentTemp > 327) ) {
      return 0;
    }
    return currentTemp;
  }
}

function initialize() ❺
```

```
{
  // Get the channel
  chanAssign = new Ant.ChannelAssignment(
    Ant.CHANNEL_TYPE_RX_NOT_TX,
    Ant.NETWORK_PLUS);
  GenericChannel.initialize(method(:onMessage), chanAssign);

  // Set the configuration
  deviceCfg = new Ant.DeviceConfig( {
    :deviceNumber => 0,                // Wildcard (any matching
                                       // device)
    :deviceType => DEVICE_TYPE,
    :transmissionType => 0,
    :messagePeriod => PERIOD,
    :radioFrequency => 57,             // ANT+ Frequency
    :searchTimeoutLowPriority => 10,   // Timeout in 25s
    :searchTimeoutHighPriority => 2,   // Timeout in 5s
    :searchThreshold => 0} );          // Pair w/all sensors
  GenericChannel.setDeviceConfig(deviceCfg);

  data = new TempeData();
  searching = true;
}

function open() ❻
{
  // Open the channel
  GenericChannel.open();

  data = new TempeData();
  searching = true;
}

function closeSensor() ❼
{
  GenericChannel.close();
}

function onMessage(msg) ❽
{
  // Parse the payload
  var payload = msg.getPayload();

  if( Ant.MSG_ID_BROADCAST_DATA == msg.messageId )
  {
    if( TempeDataPage.PAGE_NUMBER ==
        (payload[0].toNumber() & 0xFF) )
    {
      // Were we searching?
      if(searching)
      {
```

```
            searching = false;

            // Update our device configuration primarily to see the
            // device number of the sensor we paired to
            deviceCfg = GenericChannel.getDeviceConfig();
          }
          var dp = new TempeDataPage();
          dp.parse(msg.getPayload(), data);
          tempDataAvailable = true;
        }
      } // end broadcast data

      else if( Ant.MSG_ID_CHANNEL_RESPONSE_EVENT == msg.messageId )
      {
        if( Ant.MSG_ID_RF_EVENT == (payload[0] & 0xFF) )
        {
          if( Ant.MSG_CODE_EVENT_CHANNEL_CLOSED ==
              (payload[1] & 0xFF) )
          {
            open();
          }
          else if( Ant.MSG_CODE_EVENT_RX_FAIL_GO_TO_SEARCH
                   == (payload[1] & 0xFF) )
          {
            searching = true;
          }
        }
        else
        {
          //It is a channel response.
        }
      } // end channel response event

    } // end on message

}
```

NOTE

To determine the available device types, you will need to go to the ANT+ website and register for a developer account. Next, log in with your developer account, go to Downloads page on the ANT+ website, and download the PDF corresponding to the device profile you're interested in. You'll typically find the device type under the Slave Channel Configuration section of the document.

You will also find an explanation of how the data payload is structured in this document.

❶ There are several variables used throughout this program. The `DEVICE_TYPE` is the type of device with ANT+ technology we're working with, and the `PERIOD` refers to how often the sensor is expected to send new data. The other variables relate to the configuration of the radio, and the data received from it (the `data` variable).

❷ This class represents the value from the sensor that we're interested in: the current temperature.

❸ The data from the sensor comes back as *data pages*. This class reads through each page, parsing the *payload* contained in the page, and extracts the current temperature from it. The `parse()` function calls the `parseCurrentTemp()` function and stores the value it gets from it into the `currentTemp` field of the `TempeData` variable, `data`.

❹ This function receives the ANT message and extracts the current temperature from it. The ANT message comes in as an array of bytes; elements 6 and 7 are the least significant and most significant byte that, when combined, give us a two-byte integer that contains the temperature.

The range of numbers that can be represented in two bytes is 0 to 65,535. Divided by 100, that gives us a range of 0 to 655.35 degrees, in .01 degree increments. However, temperatures can be negative, so the values 0 to 32767 represent 0 to 327.67 degrees, while the values 32768 to 65535 represent the range -327.67 to 0 degrees. Technically, 0 represents positive 0 and 65535 represents negative 0, but they are numerically equivalent.

❺ This function assigns a radio channel for the ANT network, then configures the device to communicate with the tempe sensor. It then creates a new `TempeData` object and stores it in the `data` variable.

❻ The `open()` function reopens the ANT communications channel.

❼ This `closeSensor()` function closes the ANT communications channel.

❽ This method is called when a message is received from the tempe sensor. It retrieves the payload from the message, allocates a data page for parsing the payload, and parses the newly received data. It also includes some code to handle various types of activities that might occur in the course of radio communications.

After you've saved all your changes, you can run it on the device or the simulator. If you're using the simulator, you'll need the USB adapter described in "What You'll Need" on page 15 and any drivers required by it. Figure 4-6 shows the current temperature (in Celsius) on the simulator.

For further exploration of ANT and ANT+, check out the following resources:

ANT/ANT+ N5 Starter Kit
: Dynastream Innovations, a Garmin company, makes the N5 Starter Kit, which includes a programmer with cable, two ANT radios, an input/output board, USB carrier, and a battery board for powering one of the radios. You can use their N5 SDK to develop firmware that runs directly on the radio modules and create your own ANT/ANT+ products.

ANT Developer Forums
: On the ANT+ forum, you'll find discussions, advice, and answers for all your ANT/ANT+ development questions.

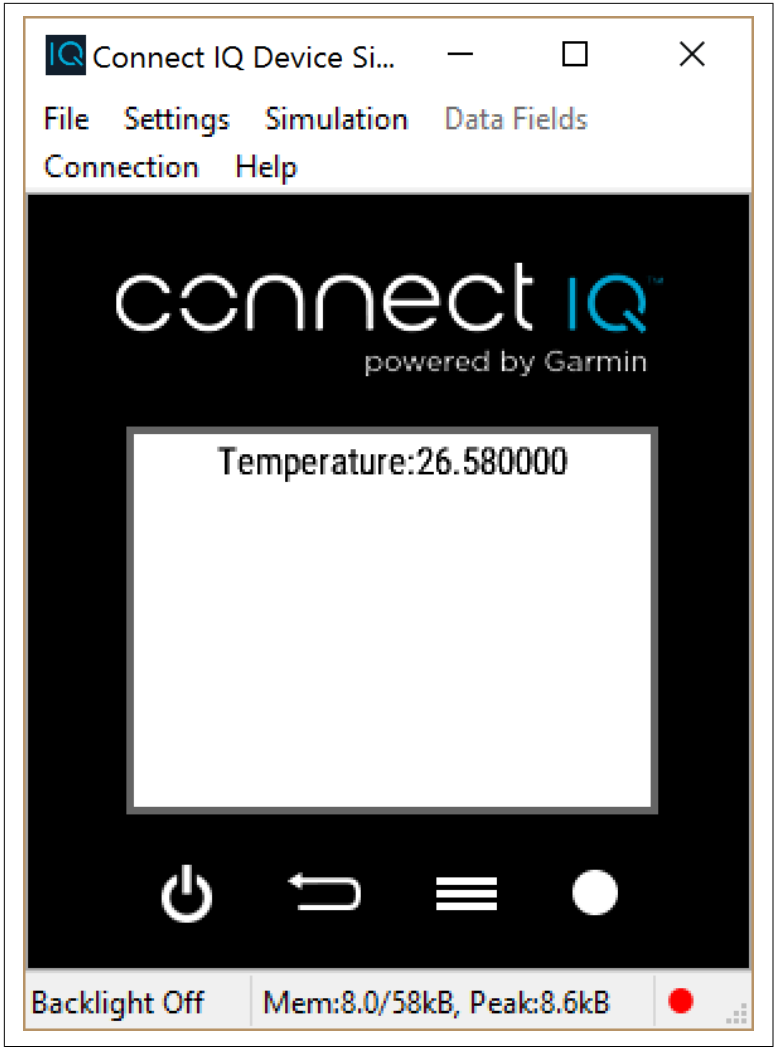

Figure 4-6. Reading the temperature

Working with an Authenticated Web Service

OAuth provides a standard way for developers to authenticate their apps against a web API. Before you can use an API in your app, you (the developer) must register yourself and your app with whoever provides the web API. Your users must also be registered as users with whoever provides the API. When your app goes to authenticate to a web API, it first uses the app's credentials, then asks the user to

log in to the service, and then gives your app permission to access her data.

OAuth provides a special challenge to wearables: how can you type your username and password into the little screen on your wrist? You can't, at least not easily. As a result, Connect IQ has a few tricks up its sleeve: first, you invoke the `makeOAuthRequest` function, which causes a notification to appear on the user's paired phone that's running the Garmin Connect app. This notification (Figure 4-7) takes the user to a web page that allows him to sign in and grant permission. Once this process is complete, the user is notified that sign-in is complete, and asks him to return to his watch.

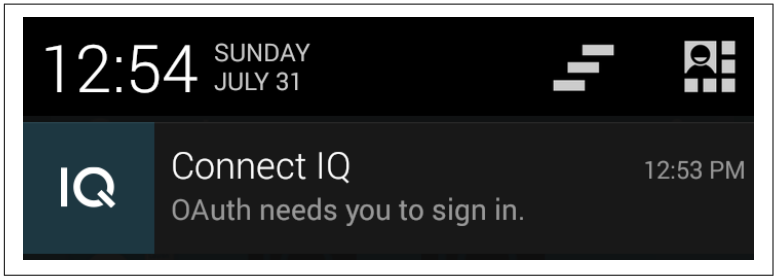

Figure 4-7. How Garmin Connect Mobile notifies users that they need to sign in

What happens if your watch app or widget times out before the user finishes logging in?

To address this, your app should call `registerForOAuthMessages` to receive the result of the login process. If your app closes before the login process completes, the result will be cached on the device until the next time your app calls `registerForOAuthMessages`, at which point the result will be passed to your callback immediately. Once you have the access token, you can use it as an argument to `makeWebRequest`, which is the function you use to actually call the API.

Activity Summary

Strava is a popular fitness app and online community. Strava can collect workout information from its own Strava app as well as other fitness communities, including Garmin Connect. Strava offers a developer API that allows access to an athlete's workout history and

other information. This section looks at an example app that accesses an athlete's four-week summary of workouts and summarizes it in a Connect IQ widget. Because it is a large app, I won't be showing it in its entirety here. The source code is accessible from GitHub.

To access the Strava API, you need to first create an app definition. To do this, you need to go to Strava's developer site and create an application. You'll need to obtain the following values from the Strava developer site, and create a *source/Keys.mc* file with the following information (replace the values in quotes with your actual ID, secret, and token):

```
var ClientId = "FROM STRAVA API ADMIN"
var ClientSecret = "FROM STRAVA API ADMIN";
var ServerToken = "FROM STRAVA API ADMIN";
```

The app is split into two halves: login and display. If the app hasn't gotten an access token, it needs to authenticate the user. Most of the work is done in the `LoginTransaction` class, which is contained in *source/StravaLogin.mc*. This class follows the OAuth login flow described earlier.

When the Strava app is launched (*source/StravaApp.mc*), its `getInitialView` method creates a `LoginView` (*StraveLoginView.mc*), which kicks off `LoginTransaction`'s `go` method, in turn calling `makeOAuthRequest`:

```
function go() {
    // Kick off a request for the user's credentials. This will
    // cause a notification from Connect Mobile to appear.
    Comm.makeOAuthRequest(
        // URL for the authorization URL
        "https://www.strava.com/oauth/authorize",
        // POST parameters
        {
            "client_id"=>$.ClientId,
             "response_type"=>"code",
             "scope"=>"public",
             "redirect_uri"=>$.RedirectUri
         },
         // Redirect URL
         $.RedirectUri,
         // Response type
         Comm.OAUTH_RESULT_TYPE_URL,
         // Value to look for
         {"code"=>"value"}
```

```
        );
    }
```

Earlier, `LoginTransaction`'s `initialize` method had set up a callback to handle OAuth messages:

```
Comm.registerForOAuthMessages(method(:accessCodeResult));
```

The `makeOAuthRequest` call will cause the login prompt to appear in Garmin Connect Mobile. Once the user has completed credential entry, the `accessCodeResult` callback is invoked with the response from the authenticating server, including the temporary access code. At this point, the user is done with the browser on her phone, and your app will use `makeWebRequest` to request the access token with an HTTP POST request. If this request is successful, it will call our delegate's `handleResponse` method with the access token:

```
// Convert the authorization code to the access token
function getAccessToken(accessCode) {
    // Request an access token via POST over HTTPS
    Comm.makeWebRequest(
        // URL
        "https://www.strava.com/oauth/token",
        // Post parameters
        {
            "client_secret"=>$.ClientSecret,
            "client_id"=>$.ClientId,
            "code"=>accessCode
        },
        // Options to the request
        {
            :method => Comm.HTTP_REQUEST_METHOD_POST
        },
        // Callback to handle response
        method(:handleAccessResponse)
    );
}

// Callback to handle receiving the access code
function handleAccessResponse(responseCode, data) {
    // If we got data back then we were successful. Otherwise
    // pass the error on to the delegate
    if( data != null) {
        _delegate.handleResponse(data);
    } else {
        Sys.println("Error in handleAccessResponse");
        Sys.println("data = " + data);
        _delegate.handleError(responseCode);
    }
}
```

What do you do with this token? Fortunately, Connect IQ offers nonvolatile storage of app properties so you won't need to reauthorize each time you run the app. For apps in the Connect IQ app store, the persistent data is encrypted using a randomly generated asymmetric key, and app access is validated with digital signatures (see the Connect IQ security model). You can use the app properties to store away the access token you receive from the server:

```
// Handle a successful response from the server
function handleResponse(data) {
    // Store the access and refresh tokens in properties
    // For app store apps the properties are encrypted using
    // a randomly generated key
    App.getApp().setProperty("refresh_token",
                             data["refresh_token"]);
    App.getApp().setProperty("access_token",
                             data["access_token"]);
    // Store away the athlete id
    App.getApp().setProperty("athlete_id",
                             data["athlete"]["id"]);
    // Switch to the data view
    Ui.switchToView(new StravaView(), null, Ui.SLIDE_IMMEDIATE);
}
```

Once you have the access token, you can use it to make authenticated requests to the server by passing the access token in the HTTP headers:

```
Comm.makeWebRequest(
    url,
    _parameters,
    {
        :method=>Comm.HTTP_REQUEST_METHOD_GET,
        :headers=>{ "Authorization"=>"Bearer " + accessToken }
    },
    method(:onResponse)
);
```

Keep It Brief

The Connect IQ Communication API brings the wearable web to Garmin devices. However, there are some subtleties to how to expose a web service to a Garmin device. Because all of the communication takes place over a Bluetooth Smart connection (you may know this as Bluetooth LE or BLE), the device is bandwidth limited. Data transfers through the Connect IQ SDK will have a transfer speed less than 1 Kb/s, generally between 400 and 800 bytes/s. For

comparison, a single tweet from Twitter's API can be upwards of 2.5 Kb. Connect IQ does some magic under the hood to minimize the amount of data that's transferred from the phone to the watch, but you can quickly see how pulling a user's last few tweets could be somewhat time consuming.

This classic proverb of "Less Is More" couldn't be more true when considering JSON responses. When working with web services, consider what information you really need to have at the Connect IQ level and keep your bandwidth budget in mind when you make decisions about what to ask a server to send back to you.

The projects in this chapter are meant to give you a foundation in Connect IQ development, but also a start on whatever you might want to build yourself. You know how to record sessions and define your own data fields. You've seen how to talk to external sensors and read their transmissions. And you have gotten an overview of using OAuth to authenticate to web services and fetch data from them.

Now you can combine these things, or mix up your own from scratch. For example, you could push your temperature readings to a web service that logs them and displays them online. You could poll a public weather API like the Weather Underground, and combine humidity and weather forecast information with your temperature readings.

The Connect IQ SDK, while preserving backward compatibility with older SDKs, is in a constant state of improvement. When I started this book, a new version was in beta; it was already on its first point release by the time I finished it. Keep an eye on the Connect IQ developer site, especially their forums and developer blog, and sign up for their developer newsletter to stay current with the latest updates.

CHAPTER 5
Our Wearable, Connected Future

If you'd asked someone prior to 2014 what a wearable technology device means to them, you'd have gotten a lot of different answers, and a smartwatch might have been one of them. By the time we reached 2015, things were taking shape, but Apple Watch was essentially an extension of an iPhone. It wasn't meant to be a standalone device; it was more of a window into the bigger world of a companion smartphone.

That's been cleaned up a lot in 2016. But even before Apple added GPS and more standalone capabilities to their watch, devices that use Connect IQ technology had been designed from the ground up to be extensions of the user rather than extensions of the user's smartphone. That's a smart place to start from, and in 2016, Apple started moving in that direction.

If you start from thinking of wearables as an extension of you, where can they go from here? The DIY, craft, and Maker movements are full of inspirational examples of wearable devices that break out of conventional definitions of wearables. Many of them blur the lines between the Internet of Things (IoT) and wearables.

The inexpensive wearable FLORA platform from Adafruit is a constant source of innovation and inspiration for anyone looking to take wearables in a new direction. Any of these projects from Becky Stern, former Director of Wearable Electronics at Adafruit, could be integrated with a wearable device with Connect IQ technology, with the addition of an ANT radio to the FLORA project, and a little bit of programming:

Sunscreen Reminder Hat

It's easy to remember to put on sunscreen when you go out, but not so easy to remember to reapply. This project uses a UV sensor to monitor how much UV you've been exposed to. You could connect this sensor to your device with Connect IQ over a wireless connection, not only to set reminders for when to put on sunblock, but to monitor, record, and graph how much UV you've been exposed to.

Brake Light Backpack

A wearable with Connect IQ technology can track a lot of things about your bike ride; this project uses an accelerometer to measure something you might not think to measure: when you step on the brakes. You could adapt the logic in this example to add a new data field to count how often you brake on a bicycle ride. You could then adapt the electronics from this project to light up when you brake (again, you'd need to establish a wireless connection between your watch and the FLORA).

NeoPixel Matrix Snowflake Sweater

Tacky holiday sweaters never go out of style, perhaps because they've always *been* out of style. This project uses a smartphone and a wireless connection to control a snowflake display on a winter sweater. Not only could you use Connect IQ to build a Watch App that controls the display, but you could map the display's behavior to your current heart rate if you so desired, pulsing at a rate that corresponds to your actual, measured heart rate.

These are the sort of places you end up when you think about the smartwatch as an extension of you, rather than a "second screen" for your smartphone. It starts right at your core, right at your heart, and it can't get any more personal than that.

A device that is so closely tied to what keeps you alive has a big responsibility, and that's to be true to you. But you can't expect a little smartwatch to be human, not without some help. As the developer, you've got the opportunity to inject humanity into the things you build. Go create things that make people healthier, happier, and more connected to one another.

About the Author

Brian Jepson is an O'Reilly editor, hacker, and co-organizer of Providence Geeks and the Rhode Island Mini Maker Faire. He's also a geek-at-large for AS220, a nonprofit arts center in Providence, Rhode Island. AS220 gives Rhode Island artists uncensored and unjuried forums for their work and also provides galleries, performance space, fabrication facilities, and live/work space.