WOLFRAM U

Building Applications with the Wolfram Cloud

Learning Objectives

In this course, you will learn to take an idea, a function or a prototype in a notebook and turn it into a working application running in the Wolfram Cloud™.

What You Will Learn

Basics of the Wolfram Cloud

Working with cloud notebooks

Working with HTTP requests and responses with APIs

Demo: Survey

Visit the following and vote:

https://wolfr.am/GSHEPnWR

After voting, you will be taken to the results, which are available at https://wolfr.am/GSHM6n2Y.

What Is the Wolfram Cloud?

The user experience for the survey is just using a website. Where does the cloud come in? There is a saying “there is no cloud, only other people’s computers.” That is a core value proposition of any cloud platform, and it is true for the Wolfram Cloud.

These are the things the Wolfram Cloud and the Wolfram Language provide:

- Server infrastructure, the Wolfram Engine™, load balancing and storage
- Much of the software for interacting with the web itself

For example, the Wolfram Cloud provides the HTML, CSS and JavaScript to implement a web form provided by **FormFunction**, and the ability to use Wolfram Notebooks™ in a web browser.

These are the things you still need to do to make a cloud application:

- Develop core application code (algorithms, visualizations)
- Deploy, monitor and manage the application using Wolfram Language functions

This course focuses on teaching you what tools you have for putting together an application and how to deploy and manage the application from the Wolfram Language.

Demystifying Cloud Applications

If you have not written applications that run on multiple web servers, or even a simple webpage, you may have ideas about how it should work but do not have a good way to think about it. Even if you do, it can help to consider the big picture before diving in.

An Application Is a Program

Your cloud-based application is just a program running on networked computers with shared storage.

What is a program?

- Takes input
- Computes something
- Produces output

The use of storage (files, databases, cloud-based storage, services returning information over a network) is just another source of input.

Concurrency (running the same program at almost the same time on different computers that share storage) is a complicating factor that cannot be ignored.

Wrapping a Core Function

Many applications have a core function that can be run by itself in the desktop application. The process of making a cloud application is one of wrapping its inputs and outputs in web-aware elements and making it work at a large scale. There can be more complex applications, of course, but they can be built in pieces from core functions.

Application Architecture

Your application's architecture is simply the system of independent programs and input/output devices and how they interact with each other.

- Simple architecture: a form that takes a sentence and returns the list of words with their parts of speech
- Complex architecture: many components, elaborate storage schema, lots of communication between components

The different tools available in the Wolfram Cloud can be combined to form applications with many types of architecture.

Life Cycle of a Cloud Object

CRUD is an industry acronym used to describe the life cycle of persistent storage:

- Create
- Read
- Update
- Delete

There are two more elements you can add:

- Inspect (find metainformation such as size and file modification time)
- Enumerate (list objects)

This acronym helps you remember everything there is to do with a cloud object. Here is a summary of functions that go with each part of the life cycle:

- Create: **CloudPut**, **CloudDeploy**, **CloudPublish**, **CloudExport**, **CopyFile**, **CopyDirectory**, **CreateDirectory**, **CloudSave**
- Read: **CloudGet**, **CloudImport**, CopyFile, CopyDirectory, **URLRead**, **URLExecute**
- Update: same as Create
- Delete: **DeleteObject**, **DeleteFile**, **DeleteDirectory**
- Inspect/Enumerate: **CloudObjects**, **RenameFile**, **Information**, **Options**, **FileExistsQ**

Remember that there are functions for working with the cloud apart from cloud objects.

CloudDirectory, CloudObject and URLs

You will now start programming with the Wolfram Cloud. Make sure you have a cloud account and are logged in with **CloudConnect**.

Find out what your current cloud directory is:

```
In[◦]:= CloudDirectory[]
```

Typically it starts in your cloud root directory:

```
In[◦]:= $CloudRootDirectory
```

Create a cloud object handle for a new directory:

```
In[◦]:= dir = CloudObject["musicsurvey"]
```

Click the hyperlink and observe what happens. You should see a page indicating the file is not found.

CloudObject is just a “handle,” a name for an object in the cloud. Evaluating CloudObject turned the relative pathname into a full URL. What you end up with is a URL “wrapped” in a CloudObject head. Other such wrappers in the Wolfram Language are **File** and **URL**. This idiom makes it easy to pattern-match on this specific type of object when writing a function definition.

Now actually create the cloud directory:

```
In[◦]:= CreateDirectory[CloudObject["musicsurvey"]]
```

Visit the link again and note the change.

List the contents of the directory:

```
In[◦]:= CloudObjects[dir]
```

The directory is initially empty, as you would expect.

Now change to this as the current cloud directory:

```
In[◦]:= SetCloudDirectory[dir]
```

Notice what happens when you construct a CloudObject now:

```
In[ ]:= CloudObject["trumpets"]
```

The relative name was added to the current cloud directory.

Notice what happens when you construct what appears to be the same CloudObject for the directory:

```
In[ ]:= CloudObject["musicsurvey"]
```

Now you see a double version of the name, because the relative name was added to the current directory. To force the name to be treated as coming from the root directory, add a “/” at the beginning:

```
In[ ]:= CloudObject["/musicsurvey"]
```

You can also build cloud object URLs using **FileNameJoin**, where the first element in the list is a directory CloudObject:

```
In[ ]:= FileNameJoin[{$CloudRootDirectory, "musicsurvey", "trumpets"}]
```

Check Your Understanding

What represents a cloud object named "report.nb" in the cloud root directory?

○ CloudObject["report.nb"]

○ CloudObject["/report.nb"]

Storing an Expression with CloudPut

Write a **DateObject** expression to a cloud object:

```
In[ ]:= date = CloudPut[Now, "date"]
```

Click the link for the object and observe how it is served. You should see a page with the **InputForm** of the expression. Most of the time, expressions written to cloud objects are intended to be used by programs, not displayed to users, so the view is to show the content as to a programmer. Note that objects are created with private permissions by default, meaning only the owning user can use them.

Read back what you wrote:

```
t = CloudGet[date]
```

Now write to the object again and read it again:

```
In[ ]:= CloudPut[Now, "date"]
        t2 = Get[date]
```

The expression stored in the object has been updated. Compare the two dates:

```
In[ ]:= {t, t2, t2 - t}
```

Deploying a FormFunction or APIFunction

Forms are a common user input idiom on the web. FormFunction provides a simple, declarative way to make one. The way you create a form is to pass it to CloudDeploy or CloudPublish.

Deploy a simple contact form:

```
In[ ]:= form = CloudPublish[FormFunction[{"Name" → String, "Email" → "EmailAddress"},
           (
             SendMail[{
               "Subject" → "New Music Survey Subscriber: " <> #Name,
               "Body" → StringTemplate["Name:``\nEmail:``"][#Name, #Email]
              }];

             "Thank you for your subscription!"
            ) &
          ],
          "contact"
         ]
```

Click the URL and try it out. It will send email to the address you logged in with.

Note that CloudDeploy and CloudPublish behave the same when two arguments are supplied, except that CloudPublish uses public permissions by default.

While **APIFunction** is intended to be used by a program instead of a human, its syntax is identical to FormFunction. You can make a contact API instead of a form just by changing the word Form-Function to APIFunction:

```
In[]:= CloudPublish[APIFunction[{"Name" → String, "Email" → "EmailAddress"},
   (
     SendMail[{
       "Subject" → "New Music Survey Subscriber: " <> #Name,
       "Body" → StringTemplate["Name:``\nEmail:``"][#Name, #Email]
      }];

     "Thank you for your subscription!"
    ) &
  ],
  "contact.api"
 ]
```

Note there is a difference between CloudPut and CloudDeploy/CloudPublish. CloudPut merely writes an expression as storage, whereas CloudDeploy and CloudPublish mark the function as a "deployment," something intended to be used or interacted with in the web environment.

Deploying a Notebook

Cloud notebooks can be created in a variety of ways. You can create a cloud notebook with code:

```
In[◦]:= nb = CloudPublish[
          Notebook[{Cell["Music Survey Results", "Title"],
            Cell["Watch this space, results will be released 6pm, " <>
              DateString[Today + Quantity[2, "Weeks"]] <> ".", "Text"]}],
          "example.nb"
         ]
```

You do not need to provide a full **Notebook** expression. The default behavior of CloudDeploy and CloudPublish is to create a notebook from a single expression. Create a bar chart:

```
In[◦]:= of = ExampleData[{"Statistics", "OldFaithful"}];
        bc = BinCounts[of⟦All, 1⟧, {{0, 2, 3, 4, 10}}];
        chart = BarChart[bc, ChartLabels → {"t < 2", "2 ≤ t < 3", "3 ≤ t < 4", "t ≥ 4"},
          PlotLabel → "Old Faithful Eruption Durations (in minutes)"]
```

Publish the chart as a cloud notebook:

```
In[◦]:= CloudPublish[chart]
```

Writing Static Cloud Objects

A static object is one that stays the same. It can be served quickly, because all the computation to produce it happened ahead of time.

CloudExport

CopyFile

Reading from Cloud Objects

- Reading a cloud object depends on its type
- Expression: read with CloudGet
- An import format: read with CloudImport
- Other formats can be downloaded with CopyFile or CopyDirectory and read locally
- An APIFunction can be called with URLRead or URLExecute

Delete, Enumeration and Metainformation

Create a directory of objects to work with:

```
In[ ]:= dir = CreateDirectory[CloudObject["meta"]]
```

Populate it with objects:

```
In[ ]:= objects = Table[CloudPut[{Now, RandomColor[]},
            FileNameJoin[{dir, "obj" <> ToString[i]}]], {i, 6}];
```

List the objects you created:

```
In[ ]:= CloudObjects[dir]
```

Delete a random object and then list the directory again:

```
In[ ]:= DeleteObject[RandomChoice[CloudObjects[dir]]]
        CloudObjects[dir]
```

Options and **SetOptions** work with cloud objects. First get a random object:

```
In[ ]:= obj = RandomChoice[CloudObjects[dir]]
```

Get its options:

```
In[ ]:= Options[obj]
```

Note that the options are actually settings stored in the cloud, so this is contacting the cloud.

Change the permissions:

```
In[ ]:= SetOptions[obj, Permissions → "Public"]
```

Get additional information about an object:

```
In[ ]:= info = Information[obj]
```

Retrieve only select fields:

```
In[ ]:= Information[obj, {"Path", "MIMEType", "FileByteCount"}]
```

Delete the entire directory:

```
In[ ]:= DeleteObject[dir]
```

You can also use DeleteFile and DeleteDirectory, but you need to make sure the object matches the type you specify or it will fail. For example, DeleteDirectory will fail if the object is not a directory. DeleteObject does not care what the object is.

Cloud Files Dashboard

You can put together the functions to create, read, update, delete, enumerate and inspect cloud objects to build a custom dashboard for managing your application.

You can create a cloud file browser that looks like this:

```
In[◦]:= Dataset[{<|"Path" → $UserURLBase <> "/", "FileType" → Directory,
           "FileByteCount" → 32 768, "LastModified" → Now|>}]
```

First write a function that extracts the information fields from an object:

```
In[◦]:= $cloudFileListingKeys = {"Path", "FileType", "FileByteCount", "LastModified"};

        cloudFileInfo[obj_CloudObject] :=
         With[{keys = $cloudFileListingKeys},
          Association[Rule @@@ Transpose[{keys, Information[obj, keys]}]]
         ]
```

Try this with an object:

```
In[◦]:= cloudFileInfo[CloudObject["/musicsurvey"]]
```

Now write a function that collects this information for the contents of a directory:

```
In[◦]:= cloudDirectoryInfo[dir_CloudObject] :=
         Map[cloudFileInfo, CloudObjects[dir]]
```

Try this with a directory, wrapping the result in **Dataset** for viewing:

```
In[◦]:= Dataset[cloudDirectoryInfo[CloudObject["/musicsurvey"]]]
```

When constructing a user interface, it is good to follow the model-view-controller paradigm. The `cloudDirectoryInfo` function gives a model of the data.

You can now write a function to present a view on the model. For example, you want to make the filename be a hyperlink with its link text as the simple filename, make the size a quantity and change the column labels. Define a function that takes the output from `cloudFileInfo` and gives an **Association** with the desired column headers and visual output for each column:

```
viewObject[info_?AssociationQ] :=
 With[{obj = CloudObject[
     "/" <> FileNameDrop[info["Path"], 1, OperatingSystem → "Unix"]]},
   <|
    "Name" → Hyperlink[FileNameTake[info["Path"], -1], First[obj]],
    "Type" → info["FileType"],
    "Size" → Quantity[info["FileByteCount"], "Bytes"],
    "Modified" → info["LastModified"]
    |>
 ]
```

In order to test it, define an overload that takes a CloudObject and fetches its info:

```
In[ ]:= viewObject[obj_CloudObject] := viewObject[cloudFileInfo[obj]]
```

Test this with an object:

```
In[ ]:= viewObject[CloudObject["/musicsurvey"]]
```

And finally, test it with directory contents:

```
In[ ]:= Dataset[Map[viewObject, cloudDirectoryInfo[CloudObject["/musicsurvey"]]]]
```

Final Cloud Files Dashboard

Finally, you want to make a complete display that updates automatically.

The dashboard will have a **DynamicModule** holding a current setting for the directory it shows and a **Dynamic** input field to edit it. It will also refresh itself every 10 seconds or on demand when the user clicks a **Refresh** button:

```
In[◦]:= cloudFileBrowser[dir_CloudObject] :=
         DynamicModule[{$refresh = 0, $dirpath = "user:" <> Information[dir, "Path"]},
          Dynamic[
           Refresh[
            Panel@Column[{
                Row[{Style["Cloud Files", Large, Bold], Spacer[200],
                  Button["Refresh", $refresh++, FrameMargins → 1]}],
                "",
                Row[{Style["Directory: ", Bold],
                  InputField[Dynamic[$dirpath], String, FieldSize → 30]}],
                Invisible[$refresh],
                If[DirectoryQ[CloudObject[$dirpath]],
                 Dataset[Map[viewObject, cloudDirectoryInfo[CloudObject[$dirpath]]]],
                 "Not a directory"
                ]
              }, BaseStyle → Directive[FontFamily → "SansSerif"]
             ],
            UpdateInterval → 10
           ],
           TrackedSymbols :→ {$refresh, $dirpath}
           (* refresh only updates to these symbols *)
          ]
         ]
```

Try this out. Note the format of the directory path, which is documented in the CloudObject reference page. You can edit the path and press Enter for it to take effect:

```
In[◦]:= cloudFileBrowser[CloudObject["/musicsurvey"]]
```

While the Dynamic interface is running, try adding a new file and a new directory:

```
In[ ]:= CloudPut[42, "data.wl"]
        CreateDirectory[CloudObject["results"]]
```

There are a couple of things to bear in mind with this interface:

- Every 10 seconds, a certain amount of information is being requested from the cloud and being sent back to your desktop application.
- The application is making a separate request for each individual cloud object.
- The Dynamic is careful to use the **TrackedSymbols** option to avoid making the desktop application unusably slow through unintended Dynamic refreshes that trigger relatively slow network calls.

The speed of your network connection to the Wolfram Cloud will play a significant role in the speed of this application. Estimate your typical connection latency:

```
In[ ]:= Median[Table[URLResponseTime[$CloudBase], {20}]]
```

Thinking about network latency and the amount of information being transferred is an important part of designing a cloud application. Think of it like being a traffic engineer for a large city. You need to know what volume of traffic to expect at what times, where there is the potential for bottlenecks, what performance metrics are important for you and how you can design for them.

Evaluation Environments

The Wolfram Cloud is divided into three evaluation environments. These environments, also known as kernel types or kernel pool types, are:

- Cloud notebook session, or simply “session” kernels
- Deployment, or public kernels
- Scheduled task, or “task” kernels

Session kernels

Deployment kernels

Task kernels

Exercising the Evaluation Environments

Check your environment and cloud evaluation status in this notebook:

```
In[ ]:= {$EvaluationEnvironment, $CloudEvaluation}
```

Whether you evaluated this notebook in the desktop application or a cloud notebook, the evaluation environment should say “Session”, and the cloud evaluation status should reflect if you are using a cloud notebook or not.

Deploy an API that reports its environment and call it:

```
In[ ]:= envapi = CloudDeploy[APIFunction[{},
            {$EvaluationEnvironment, $CloudEvaluation, $MachineName, $ProcessID} &]];
        URLExecute[envapi]
```

This does not report itself as either a “deployment” or “public” kernel. Instead it reports the specific case that it is a WebAPI, one of the uses that a deployment kernel has. If you call this API several times, you should see the machine stays the same, but the process ID changes. This illustrates the idea that the kernel is not intended to retain definitions between uses.

Deploy a dynamic notebook:

```
In[ ]:= envnb = CloudDeploy[Manipulate[{i, DateString[], $EvaluationEnvironment,
            $CloudEvaluation, $MachineName, $ProcessID}, {i, 1, 100}]]
```

Visit the link. Change the slider position to force reevaluation. In this case, the machine name and process ID both likely remain the same. This is because the system tries to route the evaluation back to the same kernel as an optimization, so it can reuse initialization settings.

Try the CloudEvaluate function:

```
In[ ]:= CloudEvaluate[
         {$EvaluationEnvironment, $CloudEvaluation, $MachineName, $ProcessID}]
```

As with the API example, this should show the same machine but different process IDs on subsequent evaluations.

Finally, try a CloudSubmit that uses the scheduled task system:

```
In[ ]:= CloudSubmit[SendMail[ToString@
           {Now, $EvaluationEnvironment, $CloudEvaluation, $MachineName, $ProcessID}],
         HandlerFunctions → <|"TaskFinished" → MessageDialog|>,
         HandlerFunctionsKeys → {"EvaluationResult"}]
```

Within a few minutes, this should send email with the results. If there is a delay, it is from a combination of the fact the background task scheduler only runs every few minutes, and other task evaluations using the available kernels.

Communication between Kernels

Cloud applications typically involve more than one kernel, whether in the same evaluation environment or not, and you will want a way to communicate from one to another. “Communication” here includes both direct, synchronous communication (A calls B) and also indirect, asynchronous communication (A leaves a message where B can find it).

Why and what would kernels communicate? Here are some possibilities:

- One kernel is acquiring and/or computing information and passing it on
- One kernel has detected an event and needs to notify another kernel to take some action

The main tools for communication between kernels are:

- Cloud objects, as shared storage
- API calls, for sending events or information from any kernel to a deployment kernel
- CloudSubmit and scheduled tasks, for triggering an evaluation in a background task

This table summarizes how one environment can affect another:

From	To Session	To Deployment	To Task
Session	cloud obj	API call, cloud obj	CloudSubmit, ScheduledTask, cloud obj
Deployment	cloud obj	API call, cloud obj	CloudSubmit, ScheduledTask, cloud obj
Task	cloud obj	API call, cloud obj	ScheduledTask, cloud obj

IncludeDefinitions

There is an automatic behavior when working with the cloud, and that is inclusion of definitions for symbols you use. This can make it easier to get started, but larger programs may want to disable this feature.

Try an example that illustrates this feature:

```
In[ ]:= PaddedMD5Sum[text_String] :=
         <|"Machine" → $MachineName,
          "Result" → IntegerString[Hash[text, "MD5"], 16, 32]|>
```

Try the function locally:

```
In[ ]:= PaddedMD5Sum[$MachineName]
```

Try the function in the cloud:

```
In[ ]:= CloudEvaluate[PaddedMD5Sum[$MachineName]]
```

In order for the expression in the CloudEvaluate to work, the definition for **`PaddedMD5Sum`** needs to be included as well.

Now try the same thing without automatic definitions:

```
In[ ]:= CloudEvaluate[PaddedMD5Sum[$MachineName], IncludeDefinitions → False]
```

Are you surprised this still worked? Notice the value of **`Machine`**. The expression was evaluated in the cloud, but since **`PaddedMD5Sum`** had no definition, it was returned unevaluated. When the result came back into the local kernel, it is evaluated one last time, but the local kernel already had a local definition of **`PaddedMD5Sum`** it was able to apply.

The typical reason to disable **IncludeDefinitions** is to separate the cloud object deployment from its code. When you have large amounts of code, it is easier to upload it as files to the cloud and instruct the cloud to load the files. This is how that might look (but do not evaluate this):

```
CloudDeploy[
 APIFunction[{"x"},
  Function[
   Needs["MyPackage`"];
   ApplyImageFilter[#x]
  ]
 ],
 IncludeDefinitions → False
]
```

In this example, even if the local kernel has a definition for `ApplyImageFilter`, it will not be included in the cloud object itself, but will instead be loaded by the **Needs** statement. All the implementation of this API has been moved to the package.

The two cases of IncludeDefinitions are analogous to static and dynamic linking in compiler technology, and the tradeoffs are similar. The advantages of automatic inclusion of definitions are:

- Lets you make sure the object you created is in sync with the definitions it uses
- The automatic behavior is simpler; there is no separate step to upload definitions and maintain them

The advantages of not including definitions:

- By separating the object from its definitions, you can independently improve the definitions
- You can share definitions between multiple objects

Exercises

1. How would you create a cloud object with a list of 100 random real numbers?

Solution

2. How could you make sure a cloud object named “report.nb” has public permissions?

Solution

3. How would you create an API that takes no arguments and returns a random number from 1 to 6?

Solution

4. How would you create an API that takes no arguments and returns a number randomly selected from a list stored in a cloud object?

Solution

5. How would you create an API that takes no arguments and returns a number randomly selected from a list provided by another API of no arguments?

Solution

6. How would you extend the cloud files dashboard to add a column with a “Delete” button that deletes the object in that row? (Bonus: add a confirmation dialog!)

Solution

Getting Notebooks into the Cloud

Besides the graphical user interface on wolframcloud.com, there are various programmatic ways to get notebooks (or other content) into the cloud.

Copy a file into the cloud:

```
In[ ]:= CopyFile["mynotebook.nb", CloudObject["test/mynotebook.nb"]]
```

Export content as a cloud notebook:

```
CloudExport[Plot[Sin[x], {x, 0, 10}], "NB", "test/myplot.nb"]
```

Deploy content (including definitions it depends on):

```
f[x_] := Sin[x];
deployed =
 CloudDeploy[Manipulate[Plot[f[a x + b], {x, 0, 6}], {{a, 2, "Multiplier"}, 1, 4},
   {{b, 0, "Phase Parameter"}, 0, 10}], "test/manipulate.nb"]
```

Publish public content:

```
published = CloudPublish[
  Manipulate[Plot[Sin[a x + b], {x, 0, 6}], {{a, 2, "Multiplier"}, 1, 4},
   {{b, 0, "Phase Parameter"}, 0, 10}], "test/published.nb"]
```

```
Out[ ]=
CloudObject[https://www.wolframcloud.com/obj/jpoeschko/test/published.nb]
```

Notice the different permissions:

```
Options[deployed, Permissions]
```

```
Out[ ]=
{Permissions → {Owner → {Read, Write, Execute}}}
```

```
Options[published, Permissions]
```

```
Out[ ]=
{Permissions → {All → {Read, Interact}, Owner → {Read, Write, Execute}}}
```

Check Your Understanding

Cloud Notebook Permissions

Permissions are specified as a mapping from users (or classes of users) to capabilities, using the cloud object option **Permissions**:

```
CloudDeploy[42, "test/perms.nb",
 Permissions → {"jfklein@wolfram.com" → {"Read", "Interact"}, All → "Read"}]
```

- `"Read"` allows viewing the notebook.
- `"Interact"` allows resolution of dynamic content and interaction (i.e. anything that needs kernel evaluations).
- `"CellEdit"` allows editing of existing cells.
- `"CellCreate"` allows creation of new cells.
- `"CellDelete"` allows deletion of cells.
- `"Evaluate"` allows evaluation of cells.
- `"Save"` allows saving of the notebook.

There are some shortcuts:

- `"Edit"` stands for `{"CellEdit", "CellCreate", "CellDelete"}`.
- `Permissions → "Public"` stands for `Permissions → {All → {"Read", "Interact"}}` (for notebooks).

Views (Cloud Object URL Types)

A cloud object can be opened in different views, corresponding to different URL types:

- The “object” view is the deployed view, usually used for publishing a notebook to a broad audience.
- The “environment” view is the editing environment in the cloud.

By default, a cloud object is referenced using its object view:

```
In[ ]:= obj = CloudPublish[Manipulate[Factor[x^n + 1], {n, 10, 100, 1}], "test/deployed"]
```

The URL type can be changed using the option **CloudObjectURLType**:

```
In[ ]:= CloudObject[obj, CloudObjectURLType → "Environment"]
```

There are several differences between views:

	Environment view	Object view
Restricted to	Authenticated users	Anyone
Supported capabilities	All	Only Read and Interact
Kernel type	Session	Public
Consuming cloud credits	No	Potentially

Check Your Understanding

Demo App: A Simple Survey

Deploy a form representing a simple survey to the cloud:

```
In[∘]:= title = "Why do you like the Wolfram Language?";
```

```
In[∘]:= prompt = "Because...";
```

```
In[∘]:= choices = {"It's good at math", "Batteries are included",
           "It has beautiful visualizations", "Everything is an expression",
           "It allows for easy deployment to the cloud"};
```

```
In[∘]:= formField[name_, caption_, choices_] :=
          {name, caption} → <|"Interpreter" → MapIndexed[#1 → First[#2] &, choices],
           "Control" → ( RadioButtonBar[##, Appearance → "Vertical"] &) |>
```

```
In[∘]:= CloudPublish[FormFunction[{formField["choice", prompt, choices]},
          "You chose answer " <> ToString[#choice] &, AppearanceRules →
           {"Title" → title, "ItemLayout" → "Vertical"}], "survey/simpleform"]
```

The chosen answer is only shown back to the user, but not stored anywhere yet.

Persisting Answers

There are various ways to persist data in the cloud:

- Using an expression in a single cloud object (dangerous without locking)
- **CloudExpression** (experimental)
- **Databin** / Wolfram Data Drop
- Connecting to another storage provider using HTTP (**URLRead**, **ServiceConnect**)
- Directly connecting to an external database (only in Enterprise Private Cloud)
- **Using one cloud object per submission** (this is the approach chosen here):

```
In[ ]:= submitChoice[targetDir_CloudObject, index_Integer] := CloudPut[Now,
          FileNameJoin[{targetDir, "choice-" <> ToString[index], CreateUUID[]}]]
```

Note the use of ***FileNameJoin*** *to create a CloudObject reference in a given (cloud) directory.*

Also note the use of ***CreateUUID*** *to create a unique random file name. The name does not really matter as long as it is unique.*

Deploy a form that persists answers using the preceding function and redirects to a (yet to be created) cloud notebook:

```
In[ ]:= CloudPublish[FormFunction[{formField["choice", prompt, choices]},
          (submitChoice[CloudObject["survey"], #choice];
            HTTPRedirect[CloudObject["survey/resultnb"]]) &,
          AppearanceRules → {"Title" → title, "ItemLayout" → "Vertical"}], "survey/form"]
```

Visualizing Results in a Notebook

Count the submitted answers and generate a **BarChart**:

```
In[∘]:= getCounts[targetDir_, choice_Integer] := Quiet[Check[Length[
            CloudObjects[FileNameJoin[{targetDir, "choice-" <> ToString[choice]}]]],
          0, {CloudObjects::cloudnf}], {CloudObjects::cloudnf}]
```

```
In[∘]:= visualizeResult[targetDir_CloudObject, choices_] :=
         BarChart[getCounts[targetDir, #] & /@ Reverse[Range[Length[choices]]],
          ChartLabels → Reverse[choices], BarOrigin → Left, ImageSize → Large]
```

```
In[∘]:= visualizeResult[CloudObject["survey"], choices]
```

Publish a notebook using this visualization to the cloud:

```
In[∘]:= CloudPublish[
         visualizeResult[CloudObject["survey"], choices], "survey/staticnb"]
```

Even if we wrapped this in **Dynamic**, the result would not necessarily update whenever a new answer is submitted (for an already open notebook), since the creation of new cloud objects is not subject to dynamic tracking.

Embedding the Notebook in a Custom HTML Page

Create an HTML page in the Wolfram Cloud that embeds the result notebook using the Wolfram Notebook Embedder library:

```
In[ ]:= getJS[nbObj_CloudObject] := StringTemplate["
        WolframNotebookEmbedder.embed(
          '`nburl`',
          document.getElementById('container')
        );
        "][<|"nburl" → First[nbObj]|>];
```

```
In[ ]:= getHTML[nbObj_CloudObject] := StringTemplate["<html>
        <head>
          <title>Survey results</title>
          <script crossorigin
               src='https://unpkg.com/wolfram-notebook-embedder@0.1/dist/wolfram-
               notebook-embedder.min.js'></script>
        </head>
        <body>
          <h1>Survey results</h1>
          <div id='container' />
          <script>`js`</script>
        </body>
        </html>"][<|"js" → getJS[nbObj]|>];
```

```
In[ ]:= CloudExport[getHTML[CloudObject["survey/staticnb"]],
         "HTML", "survey/staticresult"]
```

Controlling the Notebook from JavaScript Code

Make the dynamic content “ticklish” using a DynamicModule variable that triggers an update whenever it is changed:

```
In[ ]:= CloudPublish[DynamicModule[{counter}, Dynamic[counter;
          visualizeResult[CloudObject["survey"], choices]],
        SaveDefinitions → True, Initialization :→ (counter = 0)], "survey/resultnb"]
```

Add some JavaScript code to the outer website to change the DynamicModule variable every second:

```
getJS[nbObj_CloudObject] := StringTemplate["
WolframNotebookEmbedder.embed(
  '`nburl`',
  document.getElementById('container')
).then(function (nb) {
  nb.addEventListener('initial-render-done', function () {
    nb.getElements({}).then(function (result) {
      var id = result.elements[0].id;
      var counter = 0;
      setInterval(function () {
        nb.setDynamicModuleVariable({
          cellId: id,
          name: '$CellContext`counter$$',
          value: counter++
        });
      }, 1000);
    });
  });
});
"][<|"nburl" → First[nbObj]|>];
```

```
In[ ]:= CloudExport[getHTML[CloudObject["survey/resultnb"]], "HTML", "survey/result"]
```

Server-Side Notebook Rendering

For each notebook, the cloud generates static HTML that can be used to speed up page loading and to include the notebook’s contents in the website as seen by search engines (SEO). You can access this static HTML using the statichtml API, e.g. from an **APIFunction** rendering a website:

```
In[ ]:= getHTML[nbObj_CloudObject] :=
         Module[{nburl, statichtmlurl, statichtml}, nburl = First[nbObj];
          statichtmlurl = StringReplace[nburl, "obj" → "statichtml"];
          statichtml = URLFetch[statichtmlurl];
          StringTemplate["<html>
        <head>
        <title>Survey results</title>
        <script crossorigin
              src='https://unpkg.com/wolfram-notebook-embedder@0.1/dist/wolfram-
              notebook-embedder.min.js'></script>
        </head>
        <body>
        <h1>Survey results</h1>
        <div id='container'>`statichtml`</div>
        <script>`js`</script>
        </body>
        </html>"][<|"statichtml" → statichtml, "js" → getJS[nbObj]|>]]
```

```
In[ ]:= CloudPublish[APIFunction[{},
          getHTML[CloudObject["survey/resultnb"]] &, "HTML"], "survey/result"]
```

Putting It All Together

Combine everything into a single function that deploys the survey form, the result notebook and the APIFunction serving a webpage that embeds the notebook:

```
In[ ]:= deploySurvey[title_String, prompt_, choices_, target_CloudObject] :=
         With[{formObj = FileNameJoin[{target, "survey"}],
           nbObj = FileNameJoin[{target, "resultnb"}],
           resultObj = FileNameJoin[{target, "result"}]}, CloudPublish[FormFunction[
            {formField["choice", prompt, choices]}, (submitChoice[target, #choice];
              HTTPRedirect[resultObj]) &, AppearanceRules →
             {"Title" → title, "ItemLayout" → "Vertical"}], formObj];
          CloudPublish[DynamicModule[{counter}, Dynamic[counter;
              visualizeResult[target, choices]],
            Initialization :→ (counter = 0), SaveDefinitions → True], nbObj];
          CloudPublish[APIFunction[{}, getHTML[nbObj] &, "HTML"], resultObj];
          formObj]
```

This function still makes use of the functions `submitChoice`, `visualizeResult` and `getHTML` defined before.

Deploy a whole new survey:

```
In[ ]:= deploySurvey[
         "Which of the following European cities would you like to visit the most?",
         "I'd love to visit...", {"Barcelona", "Berlin", "London",
          "Paris", "Rome", "Vienna"}, CloudObject["citysurvey"]]
```

Exercises

1. Refactor the storage of survey submissions, using “buckets” of files (e.g. a directory for each three-character UUID prefix) instead of a single directory containing all submissions for each choice.

Hint

Solution

2. Add support for multiple questions in the survey.

Hint

Solution

3. Based on the previous code, use a **Manipulate** in the result notebook to choose which question to show the results for.

Hint

Solution

4. Based on the previous code, add buttons to the outer website’s HTML, controlling which question to show the results for.

Hint

Solution

5. Create an APIFunction that takes an expression, writes it into a cloud object (using a name based on the hash of the expression), and returns the resulting URL. Use this API to embed notebook content on a website on the fly.

Hint

Solution

Architecture

By now you know a lot about programming applications in the Wolfram Cloud. You know:

- How to use cloud objects for storage
- What the three evaluation environments are in the cloud
- How to communicate between the environments
- How to make some simple applications
- How to incorporate notebooks into your application

In this last part of the course you will learn:

- All the ways to get inputs into a cloud application
- All the ways to get output from a cloud application
- What the HTTP request/response model is
- How the Wolfram Language models HTTP requests and responses
- The debugging and management tools for working with APIs
- How a variety of example cloud applications work

Getting Information into the Cloud

From a high level, here are some options for getting information into your application:

- Wolfram Language functions that write cloud objects
    - Examples: CloudPut, CloudExport, CloudDeploy, CopyFile
    - Called from Wolfram desktop applications or **WolframScript** that runs outside the cloud
- Wolfram Language functions that read from network services
    - Examples: **EntityValue**, ServiceExecute, URLExecute
    - URLRead and URLExecute calls to external services
- Cloud objects that evaluate code in response to a web request
    - Examples: FormFunction, APIFunction, Delayed
    - These collect information from network callers via an HTTP request
- Deployed view notebooks that use user interface elements
- The Wolfram Data Drop™

Human or System Input?

The input options you choose will depend on how your application should interact with the world. Will you present a user interface to humans? Or will your application have its input from a program, as part of a larger system?

Getting Information out of the Cloud

Here are some options for getting information out of the cloud, either for humans or for systems:

- Wolfram Language functions that read cloud objects
    - Examples: CloudGet, CloudImport, CopyFile
    - Called from Wolfram desktop applications or WolframScript that runs outside the cloud
- Static cloud objects
    - Versatile: published as links, embedded in webpages, called by programs
    - Can be produced by CloudExport, generated elsewhere and uploaded to the cloud as “assets,” or produced by code running in the cloud
- Active cloud objects: FormFunction, APIFunction, Delayed
    - Output is given as the response to a request, computed from a request
    - Components outside the Wolfram Cloud can call APIs
- Deployed view notebooks
    - Output in the form of static or dynamic content
- Push data and trigger events to external services through URLRead and URLExecute

HTTP Requests

Since cloud applications use HTTP so heavily, it is worth understanding it in more depth. The Hypertext Transfer Protocol is based on a client making a request to a server, which gives a response.

Requests

HTTP Responses

The main parts of an HTTP response are:

- Status code—a three-digit number indicating the basic result of the request
  - Status codes have standard summaries and the number and summary are often written together, like 200 OK, 400 Bad Request and 404 Not Found
- Headers—a series of key/value settings that provide additional information about the response
  - Content type of the response is given in a header
  - Cookies set by the server are returned as headers
- Response body—not all responses have a body, but most do

When you use a web browser, the browser program is the client that makes a request to a web server after you type a URL in the address bar. If the server reads the response and finds the content type is HTML, the browser can start parsing the HTML syntax and it will probably find links to other resources it needs to load: JavaScript, CSS and images. The browser then makes requests to the server for those resources, and handles them accordingly to put together the complete webpage.

When you use a mobile app or a desktop app that connects to the cloud, the app code can make HTTP requests to the cloud server for a variety of things, not just webpages.

When you use web features in the Wolfram Language such as APIFunction and FormFunction, much of the HTTP process is handled automatically so you can focus on the application. But it also gives you access to these details if you want to trade off automation for more control.

HTTPRequest, URLRead and URLExecute

The **HTTPRequest** wrapper is data that represents a request. Represent a GET request to a cloud object:

```
In[ ]:= request = HTTPRequest[CloudObject["/data.json"]]
```

Access the method from the request:

```
In[ ]:= request["Method"]
```

Access both the method and URL of the request:

```
In[ ]:= request[{"Method", "URL"}]
```

Find all the available properties you can query:

```
In[ ]:= request["Properties"]
```

Access all the properties of the request as an Association:

```
In[ ]:= request[]
```

The URLRead function performs an HTTP request and returns its response.

Create the cloud object and then read it:

```
In[ ]:= dataobj = CloudExport[RandomReal[{0, 1}, 10], "JSON", "/data.json"]
        response = URLRead[HTTPRequest[CloudObject["/data.json"]]]
```

Note that URLRead provides an **HTTPResponse**. The typeset summary shows the status was `200 OK` and the content type was `application / json`, as expected.

Create an API with a simple string parameter and then call it and examine the response body:

```
In[ ]:= apiobj = CloudDeploy[APIFunction[{"x"}, StringLength[#x] &]]
        response = URLRead[HTTPRequest[apiobj, <|"Query" → {"x" → "abcde"}|>]]
        response["Body"]
```

Alter the value passed to the API's parameter:

```
In[ ]:= URLRead[HTTPRequest[apiobj, <|"Query" → {"x" → StringJoin[Alphabet[]]}|>]][
          "Body"]
```

See the HTTPRequest reference page for more information on controlling the request, including several convenience forms for specifying the request body and sending parameters such as file uploads in the request body.

URLExecute is similar to URLRead but conveniently imports the response according to the response's content type. It is also slightly more convenient to specify query parameters.

Query the API using URLExecute:

```
In[ ]:= URLExecute[apiobj, {"x" → "AB"}]
```

When the response is `200 OK`, this is convenient. Make a bad request by supplying the wrong parameter name:

```
In[ ]:= URLExecute[apiobj, {"y" → "AB"}]
```

Make a bad request by supplying a URL that does not exist:

```
In[ ]:= URLExecute[URLBuild[{$CloudBase, CreateUUID[]}], {"y" → "AB"}]
```

The response body is still imported successfully. You may want to use URLRead to get more control by definitively knowing the status code of the response, or URLExecute if detecting an error is not as important as having compact code.

Check Your Understanding

How could you represent an HTTP request
that is a GET to http://example.com/weather
with a query parameter setting of "q" to "tokyo"?

- ○ HTTPRequest["http://example.com", <|"Query"–>{"q"–>"tokyo"}|>]
- ○ URLRead[HTTPRequest["http://example.com", <|"Query"–>{"q"–>"tokyo"}|>]]

Which function would you use to call an API
that returns a JSON response and automatically
import it as a Wolfram Language data structure?

- ○ URLRead
- ○ URLExecute
- ○ HTTPRequest

HTTPResponse

The HTTPResponse wrapper represents a server response. It can be produced by URLRead, which is an actual HTTP interaction, and by **GenerateHTTPResponse**. You can also use the wrapper for your own purposes, since it is just data.

Represent a `200 OK` response with a body “Hello, world!”:

```
In[∘]:= hwresponse = HTTPResponse["Hello, world!"]
```

The typesetting box shown in the output already tells you the status code, with a green color indicator that the request is successful and the content type of the body.

Inspect the status code, content type and body:

```
In[∘]:= hwresponse[{"StatusCode", "ContentType", "Body"}]
```

List all the properties available:

```
In[∘]:= hwresponse["Properties"]
```

Retrieve all the response properties:

```
In[∘]:= hwresponse[]
```

Find the difference between this association and the list of properties:

```
In[∘]:= Complement[hwresponse["Properties"], Keys[hwresponse[]]]
```

Try each of the three body formats available:

```
In[∘]:= Grid[Map[{#, hwresponse[#]} &, {"BodyByteArray", "BodyBytes", "Body"}],
         Alignment → Left, Dividers → All]
```

The **`"BodyByteArray"`** property is the canonical representation of the body and is the one included in the association. The **`"BodyBytes"`** property gives the list of of bytes as a list of integers, and the **`"Body"`** property returns a string after applying the response’s character encoding (which is simply a way of converting between bytes and characters). Note that the body is not actually stored three ways in the HTTPResponse expression, it is only stored in the **ByteArray** format, and the other two properties convert it to a requested format.

The HTTPRequest wrapper supports the same trio of body properties, but for the request.

Represent a response with an image, noting the use of **ExportByteArray** and not **ExportString**:

```
In[ ]:= response =
         HTTPResponse[ExportByteArray[ExampleData[{"TestImage", "Aerial"}], "PNG"]]
```

Since the response is binary, the `"Body"` property that returns a string will show raw, uninterpreted bytes as characters:

```
In[ ]:= StringTake[response["Body"], 200]
```

Retrieve the body as a byte array:

```
In[ ]:= responseBytes = response["BodyByteArray"]
```

Attempt to import the image, using the content type from the response:

```
In[ ]:= ImportByteArray[responseBytes, response["ContentType"]]
```

This fails, because the response incorrectly said the content type was HTML, a text format, when in reality the response body is `image / png`. This type of inconsistency in a server response is more common than you might expect. The error was in constructing the HTTPResponse without specifying the right content type, and relying on the default. Construct it the right way:

```
In[ ]:= response = HTTPResponse[ExportByteArray[ExampleData[{"TestImage", "Aerial"}],
           "PNG"], <|"ContentType" → "image/png"|>]
```

Now the import can succeed:

```
In[ ]:= ImportByteArray[responseBytes, response["ContentType"]]
```

GenerateHTTPResponse

One tool for developing cloud applications is GenerateHTTPResponse. What does it do, and how would you use it?

- GenerateHTTPResponse works with expressions that have “active cloud object” heads, which include APIFunction, Delayed, FormFunction, **FormPage** and Delayed.
- GenerateHTTPResponse takes an active cloud object expression and an HTTPRequest and returns an HTTPResponse.
- This is exactly how the Wolfram Cloud deployment kernel serves access to a deployed APIFunction, FormFunction, etc.
- You can develop and test a lot of the behavior of an APIFunction or FormFunction without needing the cloud, using GenerateHTTPResponse, with the potential for a faster development process.

Generate a response for a simple Delayed:

```
In[◦]:= hwresponse2 = GenerateHTTPResponse[Delayed["Hello, world."]]
```

Inspect the status, body and content type for the response:

```
In[◦]:= hwresponse2[{"StatusCode", "Body", "ContentType"}]
```

Call an API with a parameter, but do not provide the parameter:

```
In[◦]:= addresponse =
         GenerateHTTPResponse[APIFunction[{"n" → "Integer"}, FactorInteger[#n] &]]
```

Inspect the body to see the error message:

```
In[◦]:= addresponse["Body"]
```

Now generate a response when the parameter is provided:

```
In[◦]:= addresponse2 = GenerateHTTPResponse[
           APIFunction[{"n" → "Integer"}, FactorInteger[#n] &],
           <|"Query" → {"n" → "16"}|>]
```

Inspect the body to see the result:

```
In[ ]:= addresponse2["Body"]
```

Note that if the body of your APIFunction uses cloud functions, GenerateHTTPResponse will go ahead and call them from your local kernel.

API Workflow

It can be helpful to look at an overview of the entire workflow of creating an API in the Wolfram Cloud. The general steps are:

- Write a core function. It may or may not use cloud features.
- Put the core function into an active head like APIFunction, Delayed or FormFunction.
- Test the APIFunction locally with GenerateHTTPResponse, to simulate HTTP request and response behavior.
- Deploy the APIFunction to a cloud object.
- Test the APIFunction with a browser or HTTP client.
- When an APIFunction is requested of the cloud, a deployment cloud kernel evaluates GenerateHTTPResponse using the deployed APIFunction and the request data (e.g. query string and request body) and the cloud turns the HTTPResponse into an actual response.

APIFunction Arguments and HTTPRequestData

It is helpful to study the way functions like APIFunction, Delayed, FormFunction and FormPage process arguments. Try an example to see the various options:

```
In[◦]:= argapi = CloudPublish[APIFunction[{
            "a",
            "b" → "Number",
            "c" → "Integer" → 0,
            "d" →  "JSON",
            "e" → DelimitedSequence[Number, ","]
          },
          {#a, #b, #c, {Length[#d], MinMax[#d], Median[#d]}, #e} &
        ]
      ]
```

Call the API:

```
In[◦]:= URLExecute[argapi,
         {"a" → "a string", "b" → "3", "d" → "[1,1,2,3,5]", "e" → "1,1,2,3,5"}]
```

You can observe several things about parameters from this example:

- Parameter types are **Interpreter** specifications.
- If a parameter name is given without a type, it defaults to **String**.
- If a type is a rule, the value is the default value, making the parameter optional.
- Parameters can be imported from known **Import** formats like JSON or PNG.
- Parameters can be imported from simple delimited sequences, like comma-separated.

Small inputs, up to a few hundred characters, can be passed in the query string of a request. Larger inputs should be passed in the request body. Try passing a large JSON array to the API:

```
In[ ]:= largedata =
          ExportString[RandomInteger[{0, 255}, 10 000], "JSON", "Compact" → True];
        URLExecute[argapi,
         {"a" → "a string", "b" → "3", "d" → largedata, "e" → "1,1,2,3,5"}]
```

You should get an error message that the URL is too long. This form of URLExecute, with parameters as a rule list, passes parameters in the query string. Instead, construct an HTTPRequest that specifies to put parameters in the body:

```
In[ ]:= largedata =
          ExportString[RandomInteger[{0, 255}, 10 000], "JSON", "Compact" → True];
        URLExecute[HTTPRequest[argapi, <|
           "Body" → {"a" → "a string", "b" → "3", "d" → largedata, "e" → "1,1,2,3,5"}|>]
        ]
```

The APIFunction automatically rejects requests where the wrong type is supplied. Try the preceding simple example, but do not provide a valid number for **b**:

```
In[ ]:= URLExecute[argapi,
         {"a" → "a string", "b" → "bad", "d" → "[1,1,2,3,5]", "e" → "1,1,2,3,5"}]
```

You should see an error message about parameter **b**. You have also seen that APIFunction automatically rejects requests that do not supply a required parameter.

You can bypass all this automation and read the raw information HTTP request, reacting in some custom way. Deploy an API with no arguments that reads **HTTPRequestData**:

```
In[ ]:= rdapi = CloudDeploy[APIFunction[{}, HTTPRequestData[] &]]
```

Call the API and inspect the data:

```
In[ ]:= URLExecute[rdapi]
```

Now call the API with some parameters, even though the API does not declare any, and notice that they show up:

```
In[ ]:= KeyTake[
         ToExpression@URLExecute[rdapi,
            {"a" → "a string", "b" → "3", "d" → "[1,1,2,3,5]", "e" → "1,1,2,3,5"}],
         {"Body", "FormRules", "Parameters", "Query", "QueryString"}
        ]
```

Create an API that computes with parameters fetched from HTTPRequestData:

```
In[ ]:= wordpatternapi =
         CloudDeploy[
          Delayed[With[{params = HTTPRequestData["Query"]},
            DictionaryLookup[Lookup[params, "pre"] ~~ ___ ~~ Lookup[params, "post"]]]]]
```

Call the API:

```
In[ ]:= URLExecute[wordpatternapi, {"pre" → "a", "post" → "ful"}]
```

The downside to this level of control is that you need to also implement error checking. This API does not check that the expected parameters are provided.

Returning Responses from APIFunction

There are several ways to control the responses from APIs and forms. By default, an APIFunction evaluates its function and the resulting expression is returned as InputForm. Deploy an API that returns a **Graphics** expression:

```
In[ ]:= wlapi = CloudPublish[APIFunction[{}, Graphics[{RandomColor[], Disk[]}] &]]
```

Click the link to visit the cloud object. Note the result is displayed in its InputForm.

Now deploy the same API but specify an export format to turn the result into an image in the JPEG format:

```
In[ ]:= wlapi =
         CloudPublish[APIFunction[{}, Graphics[{RandomColor[], Disk[]}] &, "PNG"]]
```

Read the response into the kernel, noting the response body size and content type:

```
In[ ]:= URLRead[wlapi][{"BodyByteArray", "ContentType"}]
```

What if you do not know exactly what format to return, but you will know at runtime? Deploy an API with an explicit **ExportForm** wrapper:

```
In[ ]:= wldynformat = CloudPublish[APIFunction[{"fmt" → "String" → "PNG"},
            ExportForm[Graphics[{RandomColor[], Disk[]}], #fmt] &]]
```

Call this API with different image formats:

```
In[ ]:= Map[Function[fmt,
           fmt → URLRead[HTTPRequest[wldynformat, <|"Query" → {"fmt" → fmt}|>]][
              {"BodyByteArray", "ContentType"}]],
         {"JPG", "GIF", "PNG"}]
```

What if you want an API to return a response in a format that Export does not support? This is the case where you would return an HTTPResponse expression, which lets you control not only the body content but also headers and status code:

```
In[∘]:= yamlapi = CloudPublish[
          APIFunction[{"invoice", "billto"},
           HTTPResponse[
             StringTemplate[
               "--- !example.com/^invoice
        invoice: `1`
        date   : `2`
        bill-to: &id`3`
        "][#invoice, DateString["ISODate"], #billto],
             <|"ContentType" → "application/x-yaml"|>] &
          ]]
```

```
In[∘]:= URLRead[
          HTTPRequest[yamlapi, <|"Query" → {"invoice" → "34843", "billto" → "001"}|>]][
         {"ContentType", "Body"}]
```

Debugging APIs with ResponseForm

Since APIs are usually meant to be called by a program, but programs can go wrong, how do you peek inside and see what is going wrong? Deploy an API that has some bugs in it:

```
In[∘]:= debugapi = CloudPublish[APIFunction[{"length" → "Integer"},
           RandomChoice[{heads, tails}, #length] &, {"JSON", "JSON"}]]
```

Click the link to inspect the results. There is a lot more information than without **ResponseForm**, but you should see the message that the API could not be evaluated because the required length parameter was missing.

Add the missing length parameter in a query string and visit the link:

```
In[∘]:= Hyperlink[URLBuild[First[debugapi], {"length" → "20"}]]
```

You might expect the API to work now that the length is supplied, but it still fails. This time you can see there is a message issued during the evaluation, related to JSON export. Assuming you do not immediately see the problem in the code, redeploy the API and add a **Print** statement:

```
In[∘]:= CloudPublish[
          APIFunction[{"length" → "Integer"},
           (
             data = RandomChoice[{heads, tails}, #length];
             Print[data];
             data
            ) &,
           {"JSON", "JSON"}],
          debugapi
         ];
        Hyperlink[URLBuild[First[debugapi], {"length" → "20"}]]
```

Visit the link and note the `"OutputLog"` property shows what is being sent to Export:

```
{heads, heads, heads, tails, heads, heads, heads, heads, heads, heads, heads,
tails, tails, heads, tails, heads, tails, heads, heads, tails}
```

Finally, between the Print output and the error message, you realize the bug—that these are symbols and not strings.

Deployed HTTPResponse and HTTPRedirect

How do you create a cloud object that is both static and controls every aspect of the HTTP response? Create a static cloud object that returns a YAML file with a specific cache control header:

```
In[◦]:= CloudPublish[HTTPResponse["---
         pi: 3.14159
        ", <|"ContentType" → "text/yaml",
           "Headers" → {"cache-control" → "max-age=3600"}|>],
         "/example.yaml"]
```

This combines the power and control of HTTPResponse with the speed of serving static content.

How to send a browser to a different URL computed by code? Create a form that redirects to a Wikipedia search page, which in turn serves a matching content page:

```
In[◦]:= CloudDeploy[FormFunction[
          "query" → <|"Hint" → "Search", "Label" → None|>,
          HTTPRedirect[
            URLBuild["https://en.wikipedia.org/w/index.php", {"search" → #query}]] &
         ]
        ]
```

Since redirects are usually for user interactions and not for programs, this example uses a Form-Function, but in general, many HTTP clients can follow redirects. Whether you use this depends on where you plan your cloud object to work.

Management Tools

Once an API is successful and being used by other people, you will need tools to track its usage and deal with the realities of operating an always-on program.

Deploy an example API:

```
In[ ]:= partsofspeechapi = CloudPublish[APIFunction[{"word"},
            WordData[#word, "PartsOfSpeech"] &], "/parts-of-speech"]
```

Call it at least once:

```
In[ ]:= URLRead[HTTPRequest[partsofspeechapi, <|"Query" → {"word" → "runoff"}|>]]["Body"]
```

Use **CloudLoggingData** to check the number of times it has been called and how many credits it has consumed in the last hour:

```
In[ ]:= CloudLoggingData[partsofspeechapi, "LastHour"]
```

API calls use Wolfram Cloud Credits according to the resource usage. Check your current credit balance:

```
In[ ]:= $CloudCreditsAvailable
```

However you publish your API, you may want to include a way to reach you if the API is not behaving as expected.

You may detect a problem with your API and need to temporarily shut it off while you investigate and repair it. Deactivate an API without deleting it:

```
In[ ]:= SetOptions[partsofspeechapi, "Active" → False]
```

Try accessing the API when it is deactivated:

```
In[ ]:= URLRead[partsofspeechapi]
```

Example: Accident Sign Generator

You may have seen signs like “This factory has worked 43 days without an accident,” where the sign gets updated each day without an accident event. Evaluate this code for an application that creates an electronic version of that:

```
In[◦]:= deployAccidentSignGenerator[target_String,
          units_String, description_String, permissions_ : "Public"] :=
         With[{appDir = CloudObject[target]},
          With[{indexPage = FileNameJoin[{appDir, "index.nb"}],
            resetEvent = FileNameJoin[{appDir, "reset"}],
            dataFile = FileNameJoin[{appDir, "data.wl"}]},
           Block[{$Permissions = permissions},
            Quiet[CreateDirectory[appDir]];
            CloudPut[<|"LastEvent" → Now,
              "Units" → units, "Description" → description|>, dataFile];
            CloudDeploy[Delayed[CloudPut[
               Append[CloudGet[dataFile], "LastEvent" → Now], dataFile]], resetEvent];

            CloudDeploy[
             DynamicModule[{current = CloudGet[dataFile]},
              Dynamic@Column[{
                 accidentSign[QuantityMagnitude[
                   Floor[UnitConvert[Now - current["LastEvent"], current["Units"]]]],
                  current["Units"], current["Description"]],
                 "",
                 Button["Reset Counter", URLRead[resetEvent];
                  current = CloudGet[dataFile], ImageSize → Medium]
                }], SaveDefinitions → True
             ],
             indexPage
            ];

            appDir
           ]
          ]
         ]

        accidentSign[elapsed_, units_, description_] :=
         Panel[
          Grid[{
            {""},
            {Style[elapsed, Bold, FontSize → 288, TextAlignment → Center]},
            {Style[Invisible["XYZ"], FontSize → 72]},
            {Row[{ToLowerCase[units], " ", description},
              BaseStyle → {FontSize → 48, TextAlignment → Center}]},
            {""}
           },
           BaseStyle → "Text"
          ]
         ]
```

Deploy an example application:

```
In[ ]:= deployAccidentSignGenerator["/accidentsign", "Days", "without an accident."]
```

You can also deploy an example that updates more frequently for testing:

```
In[ ]:= deployAccidentSignGenerator["/phonecompulsion",
         "Seconds", "since I checked my phone."]
```

Study the code for a few moments.

- What is the cloud object structure of the application?
- Where is input coming, and where is output going?
- What storage is used, and what is its format?

Example: My Pi Day

What are some applications that have already been built, what is their architecture, and what tools that you have seen do they use?

My Pi Day

My Pi Day (http://mypiday.com) lets users enter a date and find the spot where the date's sequence can be found within the digits of pi. The name is a tie-in to "Pi Day" (March 14 is written 3/14 in North America). How does it work?

- Based on a core function
- The core function is exposed as an API; here is an example call from HTML:

https://www.wolframcloud.com/objects/microsites/mypiday/cksum?date=10-8-19&ck=42ba5cc9
&date=10-8-19

- Since the core function can take a few seconds, and because the load can be high on Pi Day, the application caches results for a given date

Example: Pi Slices Daily and Tweet-a-Program

Pi Slices Daily

Pi Slices Daily (https://twitter.com/PiSlicesDaily) is a Twitter account that tweets the next batch of pi digits every day. How does it work?

- Based on a core function
- Computes in a scheduled task
- Sends output to Twitter
- Needs to store the next location to compute from

Tweet-a-Program

This is a Twitter account (https://twitter.com/wolframtap) that you tweet a Wolfram Language program at, and it tweets the result. How does it work?

- Tweets at the account are scanned in a WolframScript that runs forever, but could be a scheduled task.
- Individual inputs are handled by an API, called from the scanning task.
- Central task treats APIs as worker threads (in Amazon EC2, these are called lambda functions, but the Wolfram Cloud made them available earlier).

Example: Image Identify

The Image Identification Project (https://www.imageidentify.com) lets a user provide an image and the site identifies a word for what the image shows: airplane, frog, circle, etc. as a shareable link. How does it work?

- Based on a core function (**ImageIdentify**)
- Results need to be stored so they can be shared
- Results should ideally be shared so the same image input finds a previous result; can be done with a hash of the image
- Challenge: limiting the stored results
- Challenge: computationally intensive

Exercises

1. How would you represent (as data) an HTTP request with a PUT method and a JSON body?

Solution

2. How would you represent (as data) an HTTP request that set a specific user agent string?

Solution

3. How would you represent (as data) an HTTP response with a body containing a 2×2 matrix of real numbers (any numbers) in the CSV format, specifying a content type of text/csv?

Solution

4. How would you simulate a call to an APIFunction that calls FactorInteger on an integer using GenerateHTTPResponse?

Solution

5. How would you modify the wordpatternapi example to return a 400 Bad Request if the pre and post parameters were not supplied?

Solution

6. How would you extend the Accident Sign Generator example so that it remembered the dates of each event where the sign had to be reset to zero?

Solution

References

- Wolfram Cloud
- Wolfram Enterprise Private Cloud (EPC)
- Wolfram Notebook Embedder JavaScript library
- **Managing Content in the Cloud**
- **Cloud Functions and Deployment**
- **Introduction to Dynamic**