Forge - Cross-browser Add-on Development Tools

Release 1.4

Forge

April 30, 2013

CONTENTS

CHAPTER

ONE

QUICKSTART

Get started

Step 1: Sign up

Step 2: Install the Trigger Toolkit

Step 3: Read our *tutorials* and create apps

CHAPTER

TWO

FEATURES

2.1 Mobile - *see all...*

- *Address book*
- *Analytics*
- *Barcode / QR scanning*
- *Camera*
- *Child browser*
- *Cross-domain ajax*
- *Custom URL schemes*
- *Events*
- *Facebook SDK access*
- *Files*
- *Geolocation*
- *In-app payments*
- *Media*
- *Native UI enhancements*
- *Offline*
- *Push notifications*
- *Reload*
- *SMS*
- *Storage*
- *Topbar native UI*
- *Tabbar native UI*

2.2 Native plugins - *learn more...*

2.3 Browser add-ons - *see all...*

- *Background page*
- *Content scripts*
- *Cross-domain ajax*
- *Messaging*
- *Storage*
- *Toolbar button*

2.3.1 Get Started

Set up your environment

Start by reading *Getting Started with Forge* to setup your environment to interact with our cloud build service.

Then, depending on which platforms you're planning to target, to build a 'Hello World' app, you'll want to cover:

- *Forge and Mobile*
- *Forge and the Web*
- *Forge and Browser Add-ons*

Run through a tutorial

We have tutorials for mobile and browser add-ons. If you run into trouble understanding how to use the provided Forge APIs, read about *Using API methods*

- *Tutorial: creating a weather app*
- *Tutorial: creating a browser add-on which interacts with pages*

Refer to relevant recipes and API modules

Check out the full list of our *API Modules for mobile, web and browser add-ons*.

Using our own demo apps and learnings from our mobile customers over the past 6 months, we've compiled a list of *recipes and best practices* which may help you get started with building your own app.

Getting Started with Forge

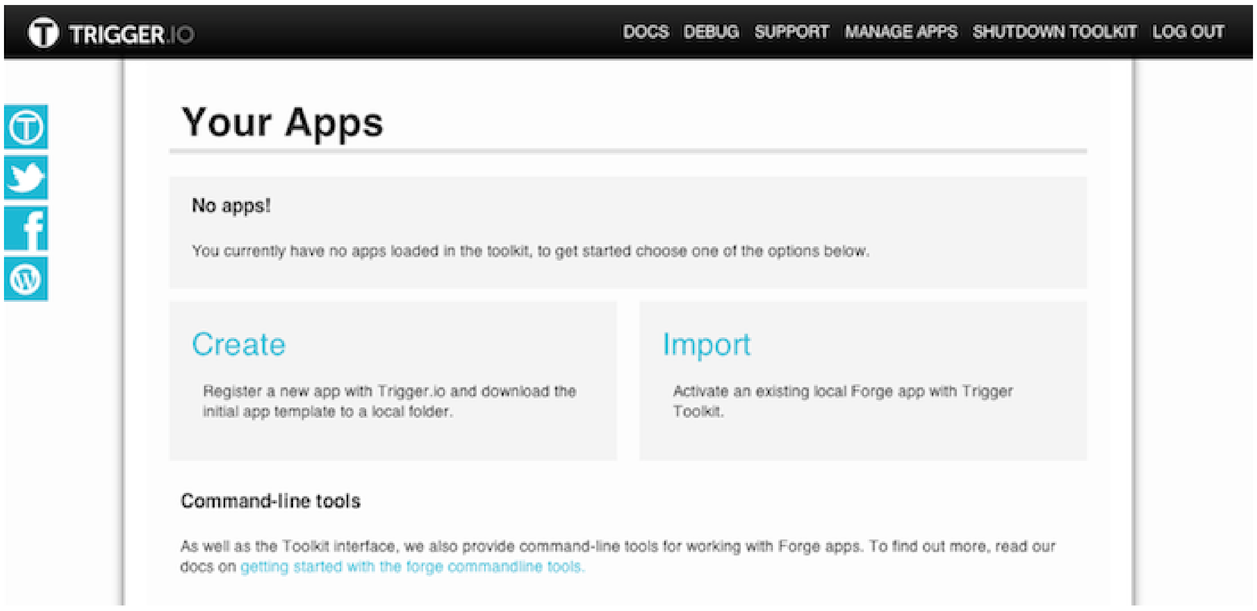
To use Forge, you will first need to install the toolkit from: https://trigger.io/forge/toolkit

After completing the install process, start the toolkit and you will be prompted to create an account. You can use the toolkit UI to build and test your apps or you can use the command-line tools that are bundled with the toolkit.

Read on to get started with the toolkit or learn about *Getting started with the command-line tools*

Getting started with the toolkit Read the instructions on the toolkit download page to learn how to install and start the toolkit on your platform.

After signing up or logging in, you will see your Your Apps. You can always return to this screen by clicking "Manage Apps" in the header. At first no apps will be listed here.



Your Apps

No apps!

You currently have no apps loaded in the toolkit, to get started choose one of the options below.

Create

Register a new app with Trigger.io and download the initial app template to a local folder.

Import

Activate an existing local Forge app with Trigger Toolkit.

Command-line tools

As well as the Toolkit interface, we also provide command-line tools for working with Forge apps. To find out more, read our docs on getting started with the forge commandline tools.

Creating your first app Click the "Create" link at the bottom-left to create a new app. You will be prompted for a name and location and then you will be returned to Your Apps where you will now see the new app listed.

Congratulations, you've created your first app and are ready to build and test it.

Working with an existing app If you have an existing app, for example you may have cloned it from an existing Github repos or created it with the command-line tools, then you will need to import it.

Simply click the "Import" link at the bottom-right of Your Apps. You will be prompted for the location and then returned to Your Apps where you will now see the app listed ready to be built and tested.

What next? By now, you have a development environment set up.

From here, you could take a look at:

- *Forge and Mobile*
- *Forge and the Web*
- *Forge and Browser Add-ons*
- *Tutorial: creating a weather app*

Getting started with the command-line tools To run forge commands use the forge executable in your Toolkit installation.

Note: It is recommended that you add the directory that the forge executable is in to your path, otherwise you will have to use the full path to the forge executable each time you want to run any `forge` command.

Windows

```
C:\> "C:\Users\<Your Username>\AppData\Local\Trigger Toolkit\forge.exe" create
```

Mac users

```
$ $HOME/Library/Trigger\ Toolkit/forge create
```

Linux users

```
$ ~/TriggerToolkit/forge create
```

Creating your first app

Note: If you have an existing app you'd like to work with, see *Working with an existing app*.

To keep each of your apps separate, we expect that you will want to work on them in different directories. In the terminal, we'll create a new directory and move into it:

```
mkdir "../demo-app"
cd "../demo-app"
```

Now, we'll create our app, with the `forge create` command:

```
$ forge create
[   INFO] Forge tools running at version 1
Enter app name:
```

At this point a descriptive name for your new app: if you're planning on following along with our tutorial, "Weather Demo" would be a reasonable choice.

If this is the first time you're running this command, you will be prompted to log in with the email address and password that you signed up with at the Forge website:

```
$ forge create
[   INFO] Forge tools running at version 2.3.1
Enter app name: Weather Demo
Your email address: james@trigger.io
Password:
[   INFO] authenticating as "james@trigger.io"
[   INFO] authentication successful
[   INFO] fetching initial project template
```

At this point, you're ready to edit your app and start running builds!

Working with an existing app If you are already working with an app on your machine, simply change directory to where the app is:

```
cd "../my-existing-app"
```

In that directory, you should have a `src` directory, containing the code for your app.

What next? By now, you have a development environment set up.

From here, you could take a look at:

- *Forge and Mobile*
- *Forge and the Web*
- *Forge and Browser Add-ons*
- *Tutorial: creating a weather app*

Forge and Mobile

The following tutorial is intended to get anyone up to speed developing iOS and Android apps using Forge.

This process does not require you to know Objective C, Java or any platform specific API calls. The only things needed is a basic understanding of HTML and JavaScript.

We're working hard to make Forge the simplest way to make mobile apps. We hope you'll find our guides easy to follow - but if you get stuck at any point, just drop us a line on support@trigger.io. We'd love to help! This guide will take you through building your first mobile app with Forge. We recommmend reading *Getting Started with Forge* first to set up your environment.

Preparing to run mobile apps Goal: Set up development environment

Forge can be used to generate both iOS and Android apps from a single codebase. You can follow the subsequent tutorial with either (or both).

If you have a Mac we recommend you get started with the iOS Simulator. With Windows or Linux, the Android emulator or device is likely to be the simplest. Of course you can also work with the Android emulator on a Mac. Or an iOS device on any development machine but that requires that you have an iOS developer account and additional setup:

- *Developing iOS apps on Windows*
- *Developing iOS apps on Linux*
- *Releasing your mobile apps*

Setting up an iOS environment To build and run iOS apps, you can use a Mac, Windows or Linux computer. However, to use the iOS simulator, a Mac is required.

To use the iOS Simulator on a Mac, you need to install Xcode. You can download Xcode from https://developer.apple.com/xcode/. When this is installed, start XCode and click 'Preferences' from the XCode menu to check the iOS Simulator and command-line tools are listed as installed under components. If not, you may install it from that window.

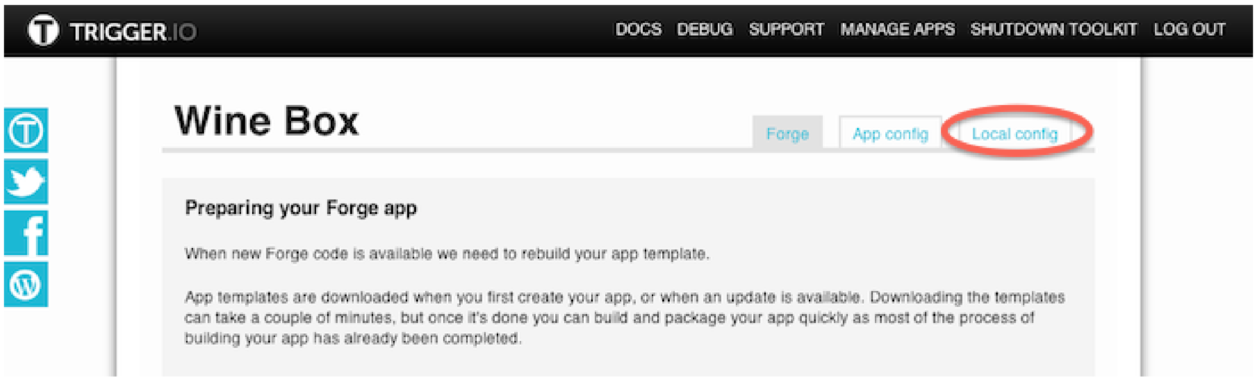
Note: you do not need to install the Xcode command-line tools just to use the iOS Simulator, but it is required for when you come to package for distribution. So we recommend you install it at the same time that you setup Xcode.

Setting up an Android environment In order to build your app for Android there is a minimum requirement of having Python and Java installed. Both commands should be installed and made available on your path.

In order to run the app, you'll need an attached phone with debug drivers installed or an active Android emulator device. You will also need to have Python and Java installed and both commands made available on your path.

If you do not set this up yourself, the tools will detect this and offer to create an appropriate Android Virtual Device (AVD) for you - simply follow the instructions given to you by the commands.

If you wish to manually manage your Android emulator you can setup the Android SDK location, by clicking on your app from the Your Apps page, then clicking on 'Local config' in the top right.



Preparing your Forge app

When new Forge code is available we need to rebuild your app template.

App templates are downloaded when you first create your app, or when an update is available. Downloading the templates can take a couple of minutes, but once it's done you can build and package your app quickly as most of the process of building your app has already been completed.

At the command-line, use the `--android.sdk` flag when using `forge run android` to point to your Android SDK location and run your own emulator AVD. All automatic installation procedures will prompt before making any changes to your system.

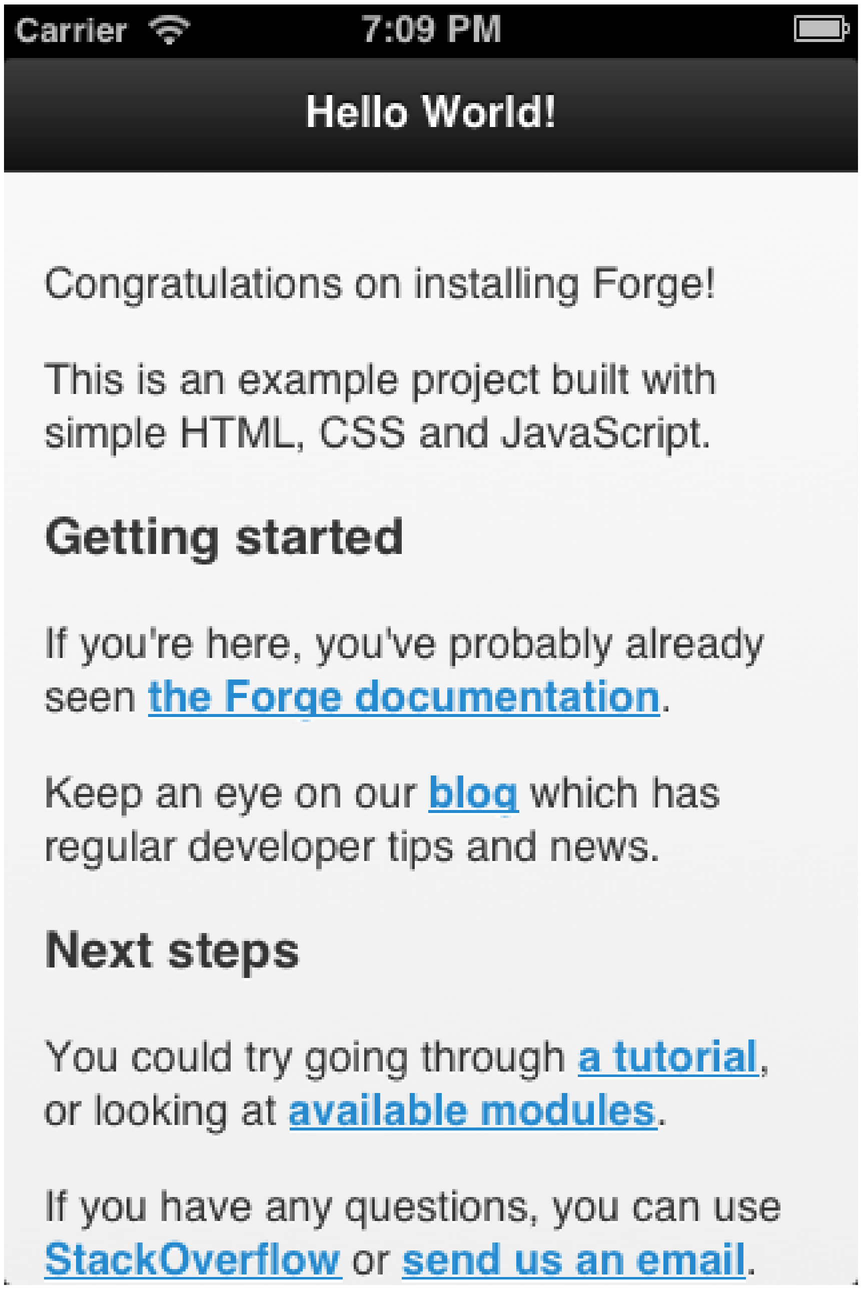
Important: There is a bug in the Android 2.3 emulator that will render your apps unusable - this is not a Trigger.io issue but if you manage your own Android AVD you should use an Android 2.2, 3 or 4+ AVD.

Building and running the code   Goal: To see your iOS and Android apps running

You can build and test your app using either the toolkit or the command-line.

Note: If you change your HTML, CSS or JavaScript between builds, then they will take just a couple of seconds. If you change the local config (in the `config.json`) file, a full rebuild will occur which will take longer

If you run a new app straight after you've created it, you will see our default 'hello world' app appear:



Hello World!

Congratulations on installing Forge!

This is an example project built with simple HTML, CSS and JavaScript.

Getting started

If you're here, you've probably already seen the Forge documentation.

Keep an eye on our blog which has regular developer tips and news.

Next steps

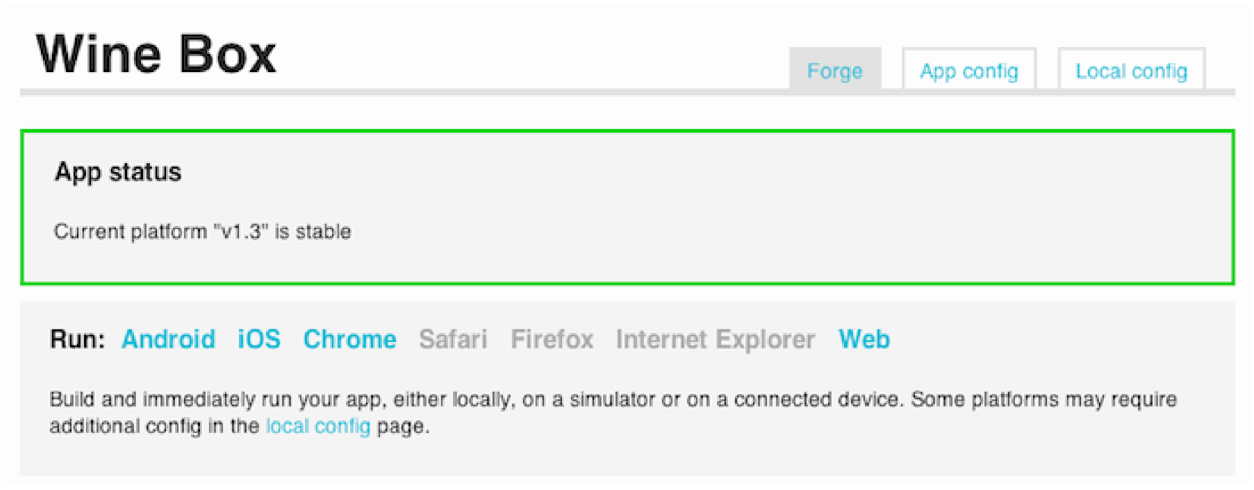
You could try going through a tutorial, or looking at available modules.

If you have any questions, you can use StackOverflow or send us an email.

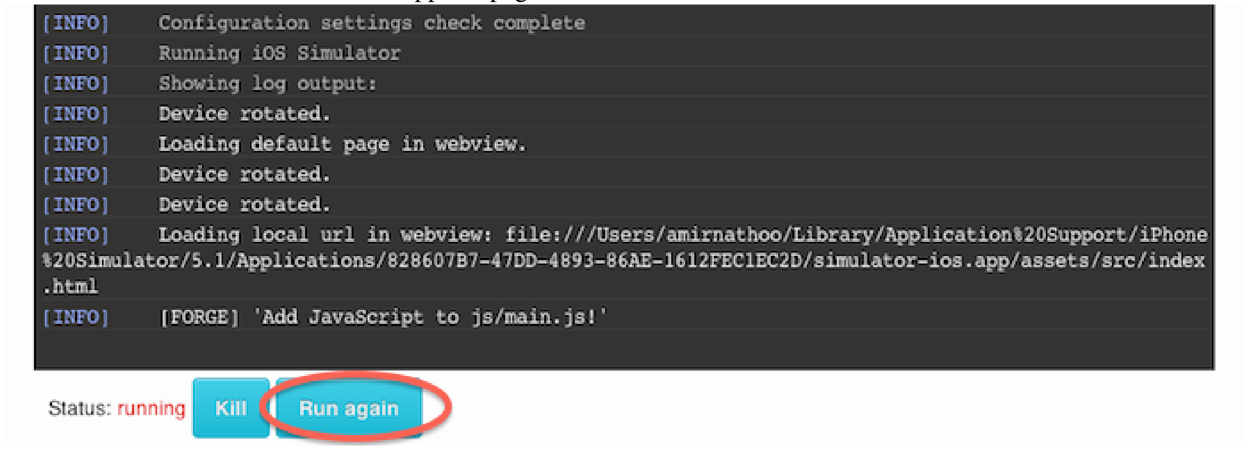
Toolkit From the Your Apps screen, simply click on the app name you wish to build.

The first time you do this, the Toolkit will perform an initial download of the resources it needs to build. This should complete in a few seconds and you will be prompted to ‘Continue’.

You can then click the appropriate link to build and run the app for Android or iOS. You will see the full traceback in the console as the commands are run so you can see progress and any warnings.


Wine Box
Forge
App config
Local config
App status
Current platform "v1.3" is stable
Run: Android iOS Chrome Safari Firefox Internet Explorer Web
Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

If you make subsequent code changes that you want to build and test on the same platform, just click 'Run again' at the bottom of the console view in the app run page.



```
[INFO]    Configuration settings check complete
[INFO]    Running iOS Simulator
[INFO]    Showing log output:
[INFO]    Device rotated.
[INFO]    Loading default page in webview.
[INFO]    Device rotated.
[INFO]    Device rotated.
[INFO]    Loading local url in webview: file:///Users/amirnathoo/Library/Application%20Support/iPhone
%20Simulator/5.1/Applications/828607B7-47DD-4893-86AE-1612FEC1EC2D/simulator-ios.app/assets/src/index
.html
[INFO]    [FORGE] 'Add JavaScript to js/main.js!'
```

Note: if you are running the app for Android using the emulator, and an AVD (Android Virtual Device) is not already started when you click the run link, it can take a long time to startup. It will be faster on subsequent runs, but in general we recommend that you develop with an Android device for a faster build / test cycle.

Command-line At the command-line you must use two commands `forge build` and `forge run` to build and test an app. See *Getting started with the command-line tools* for the location of the `forge` executable for your platform.

To build your app:

- Navigate to your app directory
- Run `forge build ios` and `forge build android` to create your iOS and Android apps.
- Whenever the `src/config.json` configuration file changes the entire app needs to be rebuilt.
- When the build finishes take a look inside the `development` directory and you should see `android` and `ios` sub-directories

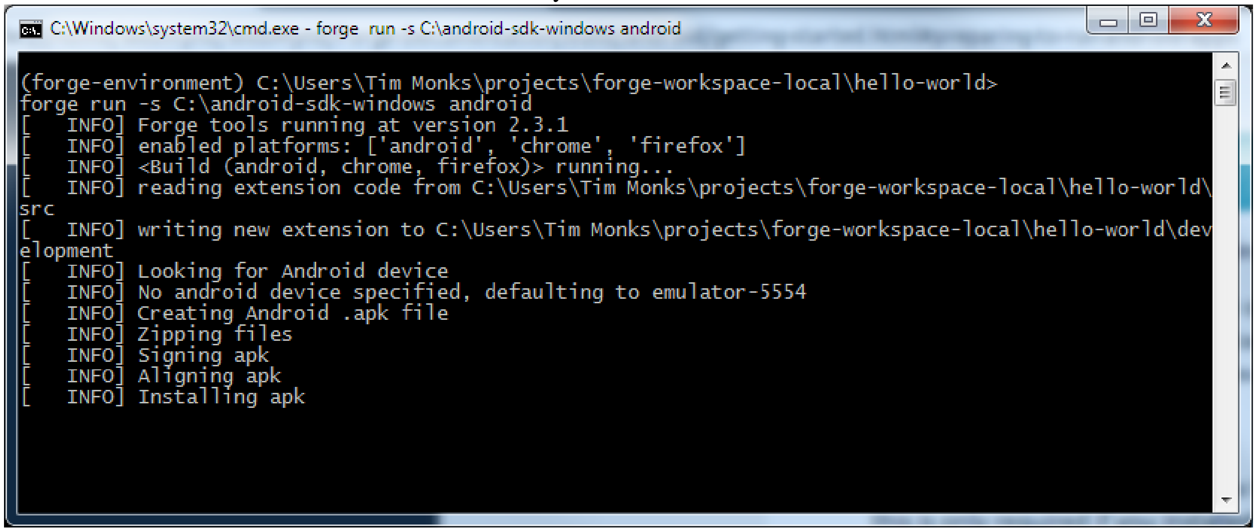
To test your app on iOS:

- Type `forge run ios`

- Apple requires apps to be packaged before deploying to iOS devices (see *releasing* for instructions) so this will launch the simulator

To test your app on Android:

- Type `forge run android`
- To use an Android device, connect it with **USB Debugging** enabled and the appropriate drivers installed
- If no device is available, we will automatically start the Android emulator



C:\Windows\system32\cmd.exe - forge run -s C:\android-sdk-windows android

```
(forge-environment) C:\Users\Tim Monks\projects\forge-workspace-local\hello-world>
forge run -s C:\android-sdk-windows android
[   INFO] Forge tools running at version 2.3.1
[   INFO] enabled platforms: ['android', 'chrome', 'firefox']
[   INFO] <Build (android, chrome, firefox)> running...
[   INFO] reading extension code from C:\Users\Tim Monks\projects\forge-workspace-local\hello-world\
src
[   INFO] writing new extension to C:\Users\Tim Monks\projects\forge-workspace-local\hello-world\dev
elopment
[   INFO] Looking for Android device
[   INFO] No android device specified, defaulting to emulator-5554
[   INFO] Creating Android .apk file
[   INFO] Zipping files
[   INFO] Signing apk
[   INFO] Aligning apk
[   INFO] Installing apk
```

Hello Mobile Goal: Adding static content

- After going through the *Getting Started with Forge* section you should see a `src` directory created inside your app directory. This is where all of your app files should be placed
- The `src` directory should contain a `config.json` file which holds all of the configuration settings for the app. You can also edit this file from inside the Toolkit, by click on your app name in the Your Apps page and then the 'App config' link in the top right.
- Open `src/index.html` in your favorite text editor
- You will see the HTML for a default "Hello World" app: you can use this as a starting point for your real app.

Note: Forge looks for index.html as the entry point of your application. **This file must be present and the name cannot be changed.**

Dynamic Hello Goal: Running dynamic JavaScript code and using logging

Let's add some dynamic functionality next and re-build:

- Replace the contents of the `body` element in `index.html` with:

  ```
  <p>Hello World, this is HTML!</p>
  ```

- Create the file `js/default.js` and change its contents to:

  ```
  forge.logging.info('Hello World, this is JavaScript');
  ```

- Open `index.html` and make sure `default.js` is being included:

```
<script type="text/javascript" src="js/default.js"></script>
```

- *Rebuild* and *re-run* the application: you should see your "Hello World" message in the app.
- Look at the command prompt/terminal running the code and you should see your "Hello World" log message.

Important: Now that you know how to use logging it is highly encouraged to use it frequently for debugging purposes.

Reference app The files in getting-started.zip represent the code you should have in your src folder at this point. If you run into any issues this is a good place to look.

It's not working! You can always contact us at support@trigger.io, or ask a question on StackOverflow.

What next? Next you could try our *Mobile Weather App* tutorial.

Or if you're comfortable with the tooling and want to do something more advanced check out *our recipes and example apps*.

Forge and the Web

In addition to creating native mobile and browser apps, Forge apps can be deployed to the open web for access by mobile and desktop browsers. This means you don't need to decide between focusing on native apps or mobile websites. Both can be created from the same HTML5 code base using the Forge framework.

Forge web apps are built upon Node.js for optimum portability. The Forge build tools handle everything required to build and deploy your code.

To hear why we created this, see Introducing 'Build to Web' - simple deployment of your mobile codebase as a web app.

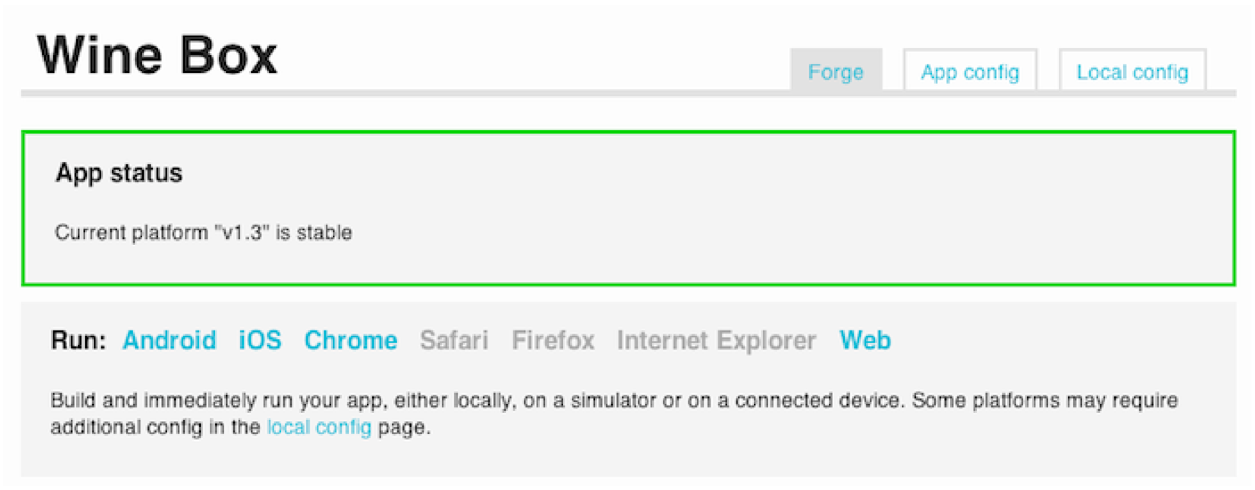
Note: Even if you don't plan to deploy to the web, this is a great place to inspect the DOM and debug your Javascript using the Chrome Developer Tools or the Firebug plugin for Firefox

Testing your web app First off you'll need a Forge application. If you're not yet set up in Forge, try our *Getting Started Guide* then our *Chrome* or *Mobile* tutorials to learn the basics - or you could download one of our demo apps from GitHub.

To run the web app locally you'll need to install node.js on your machine.

Then you can test your web app using either the toolkit or command-line tools:

Toolkit Click on the app name that you want to run from the main Your Apps page. And then just click the 'Web' link in the Run section.


Wine Box
Forge
App config
Local config
App status
Current platform "v1.3" is stable
Run: Android iOS Chrome Safari Firefox Internet Explorer Web
Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

Once the build is complete, the app will be opened up in a new tab in your default browser. It's that simple!

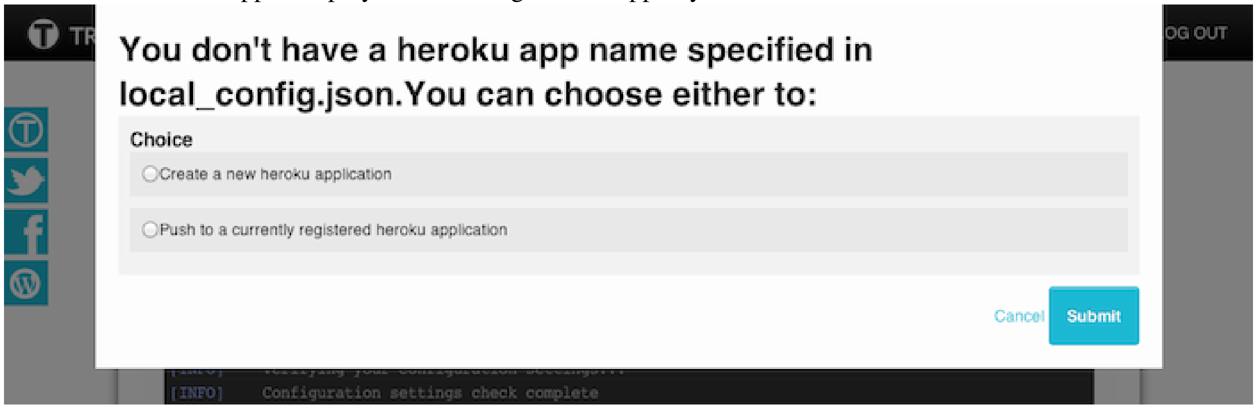
Command-line Running `forge build web` from your app's root directory creates the code for your web application alongside your mobile apps in just a few seconds (see Building the Code if this is your first time building a Forge app).

Then type `forge run web`. This will open a new tab in your browser displaying your app.

Deploying your web app on Heroku Your generated web app can be deployed to any node.js platform. To make deployment really easy, we've added an option to deploy the app directly to Heroku with a single command.

To set this you, you first need to set up a Heroku account if you haven't already. You can do this by following the steps in their tutorial. No need to deploy an application at this stage.

Toolkit Click on the app name that you want to run from the main Your Apps page. And then just click the 'Web' link in the Package section. If it is the first time you've done this for an application, you will be prompted to either create a new Heroku app or deploy to an existing Heroku app in your account.


You don't have a heroku app name specified in local_config.json.You can choose either to:

Choice
- Create a new heroku application
- Push to a currently registered heroku application

```
[INFO]    Configuration settings check complete
```

Once the build is complete, the app will be opened up in a new tab in your default browser.

That's it! Your app will be live on Heroku (the web address will be given at the command line output). You can link this to your domain name by following instructions on the Heroku site.

Command-line

- Return to your app directory and run `forge build web` to ensure your latest changes are included in the generated web app.
- Run `forge package web`. If it is the first time you've done this for an application, the command line tool will ask if you want to create a new Heroku app or deploy to an existing Heroku app in your account.

Repeat steps 2 and 3 each time you want to update the deployed app.

Deploying your web app to any node.js platform The web application lives in <your-app-folder>/release/web/heroku after you've packaged it. You can take the code from here and deploy to your favorite node.js platform like any other node.js web app.

Best practices

- Use `forge.is.web()` in the Forge Javascript framework to detect whether your app is running in a mobile browser and make any functionality or layout changes required. For example, a photo sharing app may hide native camera functionality in a web browser but show it in a mobile app.
- If you intend your output to be viewed in a desktop browser, be mindful of how it will look at this much larger screen size and design your CSS accordingly.
- Test often with `forge run web`.
- If your Node installation is in a non-standard location (e.g. homebrew installs it to `/usr/local/bin/` on OS X), you can use the `node_path` parameter to point us at the right place. Either use the "Local Config" section in the Toolkit, or edit `local_config.json` directly to add something like:

```
"web": {
    "node_path": "/usr/local/bin/
}
```

What do you think? If you have any outstanding questions or feedback we'd love to hear from you at support@trigger.io.

Forge and Browser Add-ons

The Forge platform enables you to easily and quickly write add-ons that run on Chrome, Firefox, Internet Explorer and Safari. We'll take you through how to *Get Started* below, but first lets cover some concepts:

Concepts Add-ons built on the Forge platform have two main aspects:

1. code to run when the browser is started
2. code to run when a page loads
3. code to run when a toolbar button is clicked

Code run at startup: "background code" If your add-on relies on long-running code which is not attached or associated with any particular web page, you should use background code.

This code is loaded once when the browser is open or add-on is loaded/reloaded. It runs until the browser is closed or the add-on is removed. It is good practice to put page independent logic/functionality in the background.

To use background files, you need to use the *background* module.

Code run at page load: "content scripts" If your add-on works with the individual web pages a user sees, you should use content scripts.

For example, an add-on which changes a web page so that any long words are replaced with links to that word in an online dictionary.

To use content scripts, you need to use the *activations* module.

Code run when a button is clicked: "popup code" You can use the *button* module to add an icon to the brower toolbar.

The HTML page which may be displayed when a user clicks on this icon can include JavaScript. This JavaScript is executed each time the popup is opened, and is destroyed when the popup is closed; therefore, it is closer conceptually to content scripts than background code.

Getting Started

Hello Chrome

- After going through the *Getting Started with Forge* section you should see a `src` directory created. This is where all of your add-on files will be placed.
- There will be several files and folders in the src directory: this is a basic "Hello World" app.
- Inside the folder you should see `config.json` which is automatically generated. This file holds configuration settings for your add-on. You can edit this from inside the Toolkit by click on your app from the Your Apps page, then the 'Local config' link in the top-right
- Create a file called `background.js` inside the `src/js` directory with this code:

  ```
  forge.logging.info("This is executed once per add-on/browser launch");
  ```

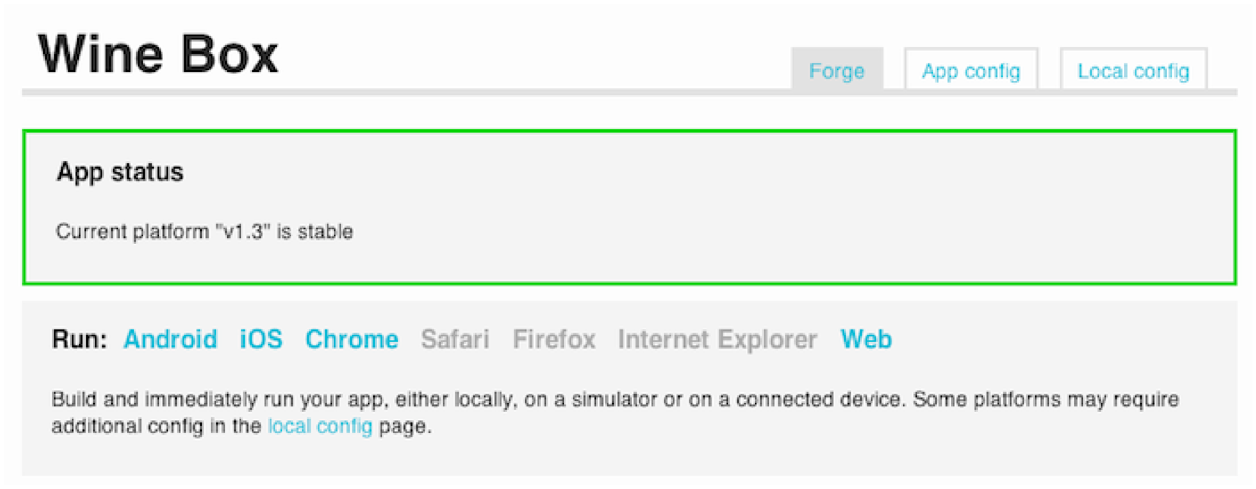
- We'll configure this to run in the *background context*, by referencing it in the configuration file.
- Open `config.json`: note how we use the *background* module to use our `background.js` file:

  ```
  "background": {
      "files": [
          "js/background.js"
      ]
  },
  ```

The next sections will explain how to build and load the add-on.

Note: Google calls add-ons on Chrome 'extensions'. Add-ons and extensions are conceptually the same thing and different platforms have different conventions on naming. When talking about Chrome specifically, we'll use 'extensions'.

Building and testing Chrome extensions using Forge To build your Chrome extension using the Toolkit, simple click on the app you wish to build from the Your Apps screen, and then the 'Chrome' link. You will see the full traceback in the console as the commands are run so you can see progress and any warnings.


Wine Box

Forge App config Local config

App status

Current platform "v1.3" is stable

Run: Android iOS Chrome Safari Firefox Internet Explorer Web

Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

If you make subsequent code changes that you want to build and test on the same platform, just click 'Run again' at the bottom of the console view in the app run page.

Using the command-line tools, use the `forge build chrome` command. When the build finishes take a look inside the `development` directory and you should see your generated Chrome extension. To test the Chrome extensions:

- Open the Chrome browser and go to `chrome:extensions`.
- If **Developer mode** isn't already enabled click the `[+]` button at the top right.
- Click **Load unpacked extension**.
- Navigate to the `development` directory which contains the generated extension.
- Select the `chrome` folder and click **OK**.
- Expand section for your Chrome extension by clicking the ?
- Click forge.html
- A Chrome debugging window will appear: this is where you can debug your background scripts.
- In the console, you should see your message: .. image:: /_static/images/developer-tools.png

It's not working! You can always contact us at support@trigger.io, or ask a question on StackOverflow.

What next? Now that you're familiar with some basics try going through the *Weather App tutorial*.

Using API methods

Most API methods provided by Forge operate asynchronously (exceptions include `forge.is` methods and `forge.tools.UUID`). This means the result of the method is not returned immediately; instead you must provide a function which will be called with the result. Care must therefore be taken when using the methods, as the order in which code is executed can be less obvious.

Here is a simplified example:

```
var url;
forge.tools.getURL("mypage.html", function (myUrl) {
    url = myUrl;
});
alert(url);
```

In this code the alert might be undefined as you cannot be sure the callback would be fired before the next line of code. Instead it would be better to say:

```
var url;
forge.tools.getURL("mypage.html", function (myUrl) {
    url = myUrl;
    alert(url);
});
```

The different callbacks which appear in Forge API methods are described below.

Callbacks Asynchronous API methods will always have the last two parameters they take as callback functions. The first of which will either be `success` or `callback` and the second will always be `error`. The way these functions are called is described below.

`success` The most commonly used callback is the `success` callback, this is the penultimate parameter to most Forge API methods. The `success` callback can be called with a variety of paramters (including none at all), depending on the particular method.

The `success` callback will only ever be called once, and will only be called if the API method completes successfully.

`callback` The `callback` callback is similar to `success` in that it appears as the penultimate parameter - however, unlike the `success` callback, it may be called multiple times. An example of its use it adding a listener to a button being clicked, as the button can be clicked multiple times the callback may be called multiple times.

`error` The `error` callback is always the final parameter passed to a method. The error callback will always be called with exactly one parameter, which will be an object containing several properties.

The returned object will always contain:

- A `message` property, which is a printable, human readable string describing the error which has occurred.

It will usually also contain:

- A `type` property, this will be one of the following strings and can be used to determine what type of error occurred and how to appropriately deal with it.
- `"BAD_INPUT"` - returned when an API is given invalid parameters, this kind of error should be eliminated before releasing your app.
- `"UNEXPECTED_FAILURE"` - returned when an unexpected error causes the API call to fail, this generally means an error in the forge API and we would be grateful if you could report this kind of error to us.
- `"EXPECTED_FAILURE"` - returned when an expected situation occurs which means the API call is not successful, examples could be a user cancelled action or a timeout. These types of errors should be handled appropriately within your application.
- `"UNAVAILABLE"` - returned when an API method is currently unavailable, this could mean unavailable on the current platform, in the current context, or at this time (i.e. if there is no Internet connection). Your application may also expected some errors of this type. Also returned for non-existant API calls.
- A `subtype` property which will give a more precise description of the error than the `type`, the strings which this may contain are documented with each API method.

It may also contain additional properties which are relevant to the API method, these properties will be documented per method in the API reference.

When errors happen the message property will always be logged by Forge, whether a callback handler exists or not.

Tutorial: creating a weather app

Tutorial Part 1 In this tutorial, we will step through building a basic weather app using the Forge tools.

This section of the tutorial will guide you through setting up the display, creating internal data representation, and doing some basic debugging using logging.

When using this tutorial, you should choose a single platform to follow along with: either iOS, Android, mobile website or Chrome extension. The code we'll be writing will work on any platform, but the configuration steps are different.

The parts that are specific to a platform will be marked with **(Mobile Only)** or **(Chrome Only)**.

All the code for this tutorial is on Github: https://github.com/trigger-corp/weather-app-demo/

Contents

Goal This part of the tutorial is intended to:

- Show how to add project files
- Set up weather forecast data structures
- Familiarize you with developing and running a basic app using Forge

Preparation

- Firstly, if you haven't already done so, go through the *getting started on mobile* or *getting started on Chrome* instructions. This will help you set up the basics and teach you how to build and run your code.
- Remove any files in the `src` directory except `config.json`, `identity.json` and the `js` folder (the other files will not be needed for the rest of this tutorial).
- Sign up for API access at http://www.wunderground.com/weather/api/ - the most basic free developer account is fine.
- Create a new javascript file called `weather.js` inside the `src/js` directory. This file will contain all of the JavaScript code for the rest of the tutorial.

- Create a file called `index.html` inside the `src` directory. This will be the html page that displays the forecast information.

Setting up the UI **Goal: Displaying static content**

Open `index.html` in your favorite editor and add the following:

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="js/weather.js"></script>
    </head>
    <body>
        Weather forecast here.
   </body>
</html>
```

Notice the script tag in the head element points to `weather.js` file you created earlier. This file does not need any code at this point.

(Chrome only) For a Chrome extension a *toolbar button* will be added near the browsers address bar which will display `index.html` when clicked.

Click on your App Config tab in the Toolkit or edit `config.json` to add the following configuration to the `modules` section.

Important: JSON requires all key value pairs to be separated by commas. Makes sure to place proceeding or trailing commas as appropriate!

```
"button": {
    "default_popup": "index.html"
},
```

Build and run the code

- Instruction on how to build and run a mobile app can be found *here*.
- Instructions on how to build and load an extension for Chrome can be found *here*.

On Chrome, a new toolbar icon should be visible!

Create dummy data **Goal: Set up some dummy data for a weather forecast** First, we will create some dummy data in a simplified version of the format that the Wunderground API will return to us - open `src/js/weather.js` and paste the following code:

```
var weather = {
    "current_observation": {
        "display_location": {
            "full": "San Francisco, CA"
        },
        "observation_time":"Last Updated on September 20, 3:50 AM PDT",
        "weather": "Mostly Cloudy",
        "temp_f": 54.4,
        "temp_c": 12.4,
        "relative_humidity":"89%",
        "wind_string":"From the WNW at 4.0 MPH",
        "icon_url":"http://icons-ak.wxug.com/i/c/k/nt_mostlycloudy.gif"
    },
    "forecast": {
        "simpleforecast": {
            "forecastday": [
                { "date": { "weekday_short": "Thu" },
                  "period": 1,
                  "high": { "fahrenheit": "64", "celsius": "18" },
                  "low": { "fahrenheit": "54", "celsius": "12" },
                  "conditions": "Partly Cloudy",
                  "icon_url":"http://icons-ak.wxug.com/i/c/k/partlycloudy.gif" },
                { "date": { "weekday_short": "Fri" },
                  "period": 2,
                  "high": { "fahrenheit": "70", "celsius": "21" },
                  "low": { "fahrenheit": "54", "celsius": "12" },
                  "conditions": "Mostly Cloudy",
                  "icon_url":"http://icons-ak.wxug.com/i/c/k/mostlycloudy.gif" },
                { "date": { "weekday_short": "Sat" },
                  "period": 3,
                  "high": { "fahrenheit": "70", "celsius": "21" },
                  "low": { "fahrenheit": "52", "celsius": "11" },
                  "conditions": "Partly Cloudy",
                  "icon_url":"http://icons-ak.wxug.com/i/c/k/partlycloudy.gif" }
            ]
        }
    }
};
```

Check the data   Goal: Confirm our data has been correctly populated by using logging

If we need to verify that our app is showing the right forecast in the future, it would be useful to log out what data input is. We can use the logging module for this.

Add this to the end of `js/weather.js`:

```
forge.logging.info(JSON.stringify(weather));
```

Remote Debugging on Mobile   Goal getting started with Catalyst

As you've already seen in *getting started on mobile* `forge.logging.info` prints output to console/terminal. You can also use our remote debugging tool, Catalyst, which provides some helpful tools for troubleshooting and examining the app at runtime.

If you're working with Chrome, you can just use the Chrome Developer tools by right-clicking on the popup: see the next section.

For a screencast on Catalyst, and help on how to get started see Screencast: Trigger.io Catalyst in action.

1. Open up a browser and go to http://trigger.io/catalyst/.
2. On this page there will be a generated `script` tag which you copy and insert into the head element of your `index.html` file.
3. Click on the auto-generated link which takes you to a page that looks similar to Chrome's debugging tools.
4. Open `src/js/weather.js` and add the following at the **beginning** of the file:

```
window.forge.enableDebug();
```

This will ensure that Catalyst is connected and ready before the code runs, preventing any logging from being lost.

5. Rebuild and re-run your app. In a few moments, your Catalyst tab in the browser should show the device.
6. Check the console of the Catalyst tool: you should see your forecast object being logged.

Note: Catalyst is a great tool, especially for debugging mobile apps: check out the "Elements" view to inspect and modify the DOM, the "API" tab to see your `forge` calls flowing back and forth, and the "Network" view to diagnose performance problems.

Debugging on Chrome **Goal: Checking forge.logging.log output in Chrome console**

`forge.logging.log` output can be seen in the Chrome console. Since `weather.js` is running inside `index.html` we need to inspect that page to see the logged output.

- Open up a Chrome browser and go to chrome:extensions
- If you have already added your Chrome extension, refresh it (Chrome caches aggressively - refreshing a few times is a good idea)
- If you haven't added your Chrome extension yet, see *loading an extension in Chrome*
- Open your app's popup by clicking the toolbar button, right-click and pick **Inspect pop-up**
- This will open up the Chrome developer tools for your popup in a new window
- At the bottom is the console section, which should contain the output from `forge.logging.log`
- Inspect the logged properties of the forecast object and make sure everything looks OK

The *background* context also receives the logging call for debugging convenience.

- Navigate to chrome:extensions
- You should see a **Inspect active views** option, with a `forge.html` link
- Click `forge.html` which will open up the Chrome developer tools for your background page
- The console may not be displayed automatically, but it can be opened by pressing the Esc key or clicking the console button on the bottom left

Reference app See the `part-1` tag in the Github repository for a reference app for this stage of the tutorial.

part-1.zip

What next? Continue on to *Tutorial Part 2*!

Tutorial Part 2 In this part of the tutorial, we'll explore one way to take the raw weather data and display it in a HTML interface.

Contents

Goal This part of the tutorial is intended to:

- Embed external scripts
- Display dynamic data using jQuery and Mustache templating

Adding external libraries **Goal: Embedding jQuery and Mustache libraries**

jQuery provides a lot of helpful functionality, and in order to display the forecast information we will use Mustache Templates.

Download the jQuery and Mustache from their project pages and save them in the `js` directory.

To use these libraries in your app simply append the following script tags in the head of `index.html` before the `js/weather.js` script tag:

```
<script type="text/javascript" src="js/jquery.min.js"></script>
<script type="text/javascript" src="js/mustache.min.js"></script>
```

It's that simple. You can now access jQuery using "$" or "jQuery" inside `index.html` and `weather.js`.

Displaying the Data **Goal: Displaying dynamic data**

Using Mustache, it is quite simple to display the data.

Note: To prevent attempts to parse Mustache or other templates as HTML, we recommend wrapping them in script tags with custom type attributes

- Open `index.html`
- Remove "Weather forecast here." from the body tag
- Append to the body tag a Mustache template to represent the *forecast information*:

```
<script type="x-mustache-template" id="forecast_information_tmpl">
    <h1>Forecast for {{display_location.full}}</h1>
    <p>{{observation_time}}</p>
</script>
```

- Next we need a template to render the *current conditions* object:

```
<script type="x-mustache-template" id="current_conditions_tmpl">
    <table>
        <tr>
            <td><img src="{{icon_url}}" /></td>
            <td>
                <div>{{weather}}</div>
                <div>{{temp_f}}&deg;F</div>
                <div>Humidity: {{relative_humidity}}</div>
                <div>Wind: {{wind_string}}</div>
            </td>
        </tr>
    </table>
</script>
```

- And finally add a template for the daily forecast data. Here, we're using Mustache's Enumerable syntax to loop through a few days' conditions:

```
<script type="x-mustache-template" id="forecast_conditions_tmpl">
    {{#forecastday}}
    <td>
        <h2>{{date.weekday_short}}</h2>
        <img src="{{icon_url}}">
        <div>{{conditions}}</div>
        <div>Low: {{low.fahrenheit}}&deg;F</div>
        <div>High: {{high.fahrenheit}}&deg;F</div>
    </td>
    {{/forecastday}}
</script>
```

- Next we need designated elements where the templated information will be appended. Add the following tags following the templates inside the body element:

```
<header id="forecast_information"></header>

<section id="current_conditions"></section>

<section id="forecast_conditions">
    <table>
        <tr>
        </tr>
    </table>
</section>
```

- Now open `weather.js` and add the following JavaScript code which will template and append the data:

```
function populateWeatherConditions (weather) {
    var tmpl, output;
    forge.logging.log("[populateWeatherConditions] beginning populating weather conditions");

    tmpl = $("#forecast_information_tmpl").html();
    output = Mustache.to_html(tmpl, weather.current_observation);
    $("#forecast_information").append(output);
    forge.logging.log("[populateWeatherConditions] finished populating forecast information");

    tmpl = $("#current_conditions_tmpl").html();
    output = Mustache.to_html(tmpl, weather.current_observation);
    $("#current_conditions").append(output);
    forge.logging.log("[populateWeatherConditions] finished populating current conditions");

    tmpl = $("#forecast_conditions_tmpl").html();
    output = Mustache.to_html(tmpl, weather.forecast.simpleforecast);
    $("#forecast_conditions table tr").append(output);
    forge.logging.log("[populateWeatherConditions] finished populating forecast conditions");

    forge.logging.log("[populateWeatherConditions] finished populating weather conditions");
};
```

- Finally add a jQuery.ready listener inside `weather.js` which will kick things off when the page finishes loading:

```
$(function () {
    populateWeatherConditions(weather);
});
```

Important: Any code that modifies the page should only be run when the page is finished loading. The above achieves this using jQuery's document ready listener `$(function () { /* code here */ })`.

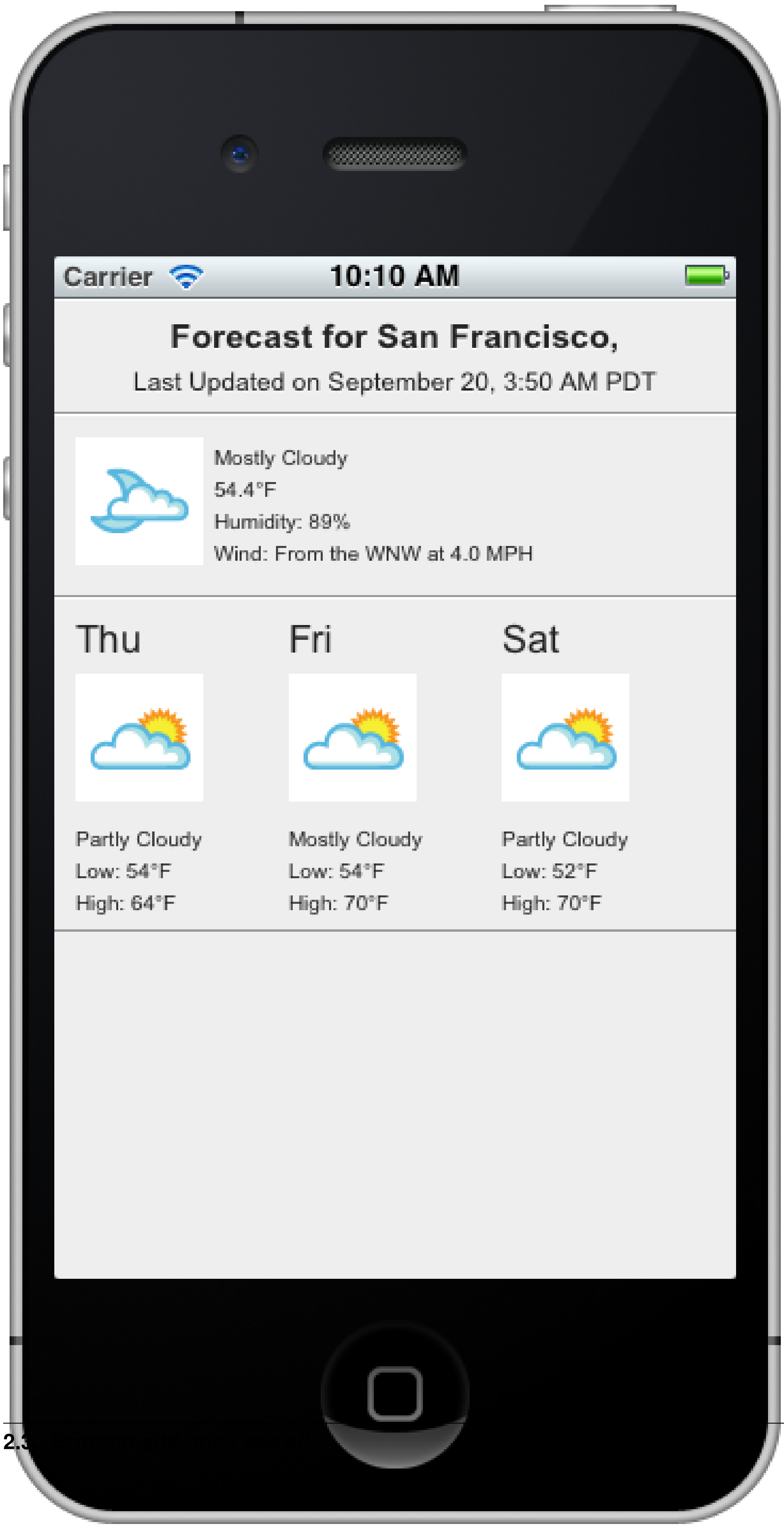
(Mobile Only) *Build* the code and *run* the app and you should see the dummy weather forecast displayed automatically.

(Chrome Only) *Build* the code and *reload* the extension. When you click on the toolbar button you should see the weather forecast displayed in a pop-up window.

Adding CSS You can make the display a bit more pleasant by adding some custom CSS. Create a `css` directory and download style.css into it. Link this file in the head element of `index.html` to add some basic styling to the Weather App:

```
<link rel="stylesheet" type="text/css" href="css/style.css">
```

At this point, your app should display static weather data for San Francisco, CA when it is opened.



Forecast for San Francisco,

Last Updated on September 20, 3:50 AM PDT

Mostly Cloudy
54.4°F
Humidity: 89%
Wind: From the WNW at 4.0 MPH

Thu	Fri	Sat
Partly Cloudy	Mostly Cloudy	Partly Cloudy
Low: 54°F	Low: 54°F	Low: 52°F
High: 64°F	High: 70°F	High: 70°F

Reference app See the `part-2` tag in the Github repository for a reference app for this stage of the tutorial.

part-2.zip

What next? Continue on to *Tutorial Part 3*!

Tutorial Part 3 This part of the tutorial will demonstrate how to use the Forge *request* module to retrieve data from the Wunderground API. We will then parse the data to pull the necessary information to generate our internal representation. We will then parse the data to pull the necessary information to generate our internal representation.

Contents

Goal This part of the tutorial is intended to:

- Show how to do cross-domain requests
- Using forge.request.ajax
- Fetch and display live data

Remove Test Dummy Code **Goal: Remove irrelevant code**

In this part of the tutorial we are going to be fetching live data, so we can remove the dummy data. Open `weather.js` and ...

- Remove the `var weather = { ... }` declaration
- Remove the `forge.logging.log(weather)` call
- Remove `populateWeatherConditions` invocation from the document ready listener (Note you do not need to remove the entire jQuery document ready listener as it will be used again in the following sections)

Understanding the Data **Goal: Gain an understanding of what data is available and how it's structured**

If you have not done so already sign up for API access at http://www.wunderground.com/weather/api/ and make a note of your API key.

To request a forecast which includes current conditions from the Weather Underground API you will specify an URL using the following format:

http://api.wunderground.com/api/API_KEY/conditions/forecast/q/STATE/CITY_NAME.json

Replace `API_KEY` by your Weather Underground API key, `STATE` by the two-letter abbreviation of the city's state and `CITY_NAME` by a city name with any spaces replaced by an underscore (_).

For example to look up weather in New York City, you could use:

http://api.wunderground.com/api/API_KEY/conditions/forecast/q/NY/New_York.json

The data returned will be JSON formatted with an identical structure as our sample data from the first tutorial though with much more detail.

Our `forecast_information_tmpl` template is populated from the `current_observation.display_location` and `current_observation.observation_time` sections:

```
...
"current_observation": {
    ...
    "display_location": {
        "full":"New York, NY",
        "latitude":"40.75013351",
        "longitude":"-73.99700928",
        "elevation":"17.00000000"
    },
    "station_id":"KNYNEWYO62",
    "observation_time":"Last Updated on September 21, 4:30 AM EDT",
    ...
```

The `current_conditions_tmpl` template is populated from variables in the `current_observation` section:

```
...
"current_observation": {
    ...
    "weather":"Mostly Cloudy",
    "temp_f":63.7,
    "temp_c":17.6,
    "relative_humidity":"74%",
    "wind_string":"Calm",
    "icon_url":"http://icons-ak.wxug.com/i/c/k/nt_mostlycloudy.gif",
    ...
```

Finally, the `forecast_conditions_tmpl` template is populated from variables in the `current_observation.forecast.simple_forecast` section:

```
...
"current_observation": {
    ...
},
"forecast":{
    "txt_forecast": {
    ...
    },
    "simpleforecast": {
        "forecastday": [
            {"date":{
                "weekday_short":"Fri",
            },
             "period":1,
             "high": {
                 "fahrenheit":"72",
                 "celsius":"22"
             },
             "low": {
                 "fahrenheit":"64",
                 "celsius":"18"
             },
             ...
```

Adding Permissions Since we are retrieving data from a 3rd party, we need to enable the *request* module and list the URLs we want to access at run-time.

Either look at the App Config section in the Toolkit, or edit `config.json` to add this request module configuration to the `modules` object:

```
"requests": {
    "permissions": ["http://api.wunderground.com/api/*"]
}
```

The items in the `permissions` array are match patterns: see http://code.google.com/chrome/extensions/match_patterns.html.

The next time you build, re-creating your app will take longer than usual: changing the configuration of your app means we need to do some work server-side.

Fetching Data **Goal: Using forge.request.ajax**

Now that you have a feel for what the returned data looks like, let's add a function to `weather.js` that will retrieve this data:

```
function getWeatherInfo(location) {
    var api_key = "YOUR_API_KEY";
    forge.logging.info("[getWeatherInfo] getting weather for for " + location);
    forge.request.ajax({
        url: "http://api.wunderground.com/api/" + api_key +
               "/conditions/forecast/q/" + location + ".json",
        dataType: "json",
        success: function (data) {
            forge.logging.info("[getWeatherInfo] success");
        },
        error: function (error) {
            forge.logging.error("[getWeatherInfo] " + JSON.stringify(error));
        }
    });
};
```

`forge.request.ajax` is similar to the behaviour of jQuery's `$.ajax`, where we specify the url, dataType to be returned, success and error callbacks.

The returned data is a Document object which can be easily parsed with jQuery.

Remember to specify your weather underground API key or the API request will fail!

```
var api_key = "YOUR_API_KEY";
```

At this point the function doesn't actually do anything with the data but you can test to see if the ajax call succeeded. For example to look up the forecast in San Francisco add the following code to the document ready listener:

```
$(function() {
    getWeatherInfo("CA/San_Francisco");
});
```

You can verify that this call is working by checking the console output. Expect to see log output like:

```
[FORGE] '[getWeatherInfo] getting weather for for CA/San_Francisco'
[FORGE] '[getWeatherInfo] success'
```

- **(Mobile Only)** Check either the command prompt/terminal or console of *Catalyst*
- **(Chrome Only)** Check the console of the *pop-up*

Fetching and Populating Data for a US City Alter the `getWeatherInfo` function to take an extra callback parameter that will be called if the retrieval was successful. The code should now look like:

```
function getWeatherInfo(location, callback) {
    var api_key = "YOUR_API_KEY";
    forge.logging.info("[getWeatherInfo] getting weather for for " + location);
    forge.request.ajax({
        url: "http://api.wunderground.com/api/" + api_key +
                "/conditions/forecast/q/" + location + ".json",
        dataType: "json",
        success: function (data) {
            forge.logging.info("[getWeatherInfo] success");
            callback(data);
        },
        error: function (error) {
            forge.logging.error("[getWeatherInfo] " + JSON.stringify(error));
        }
    });
};
```

Since we already have a function to populate the GUI we just pass that in as the callback to `getWeatherInfo`.The new call would look like:

```
$(function(){
    getWeatherInfo("CA/San_Francisco", populateWeatherConditions);
});
```

Rebuild and run the code to see live forecast data displayed.

Reference app See the `part-3` tag in the Github repository for a reference app for this stage of the tutorial.

part-3.zip

What next? Continue on to the last part: *Tutorial Part 4*!

Tutorial Part 4 In this part of the tutorial we'll add some UI components that allow the user to select a city and use persistent storage to retrieve the information when the application restarts.

Goal This part of the tutorial is intended to:

- Show how to use persistent storage
- Set up city selection UI
- Add listeners to handle city selection changes

Modyfying the UI **Goal: Add city selection UI**

We need to add a drop down which will allow the user to select and get forecast information for a particular city. Open `index.html` and append the following to the body element:

```
<div id="options">
    <div id="options_header">Select City</div>
    <select id="city_menu"></select>
</div>
```

Populating City Selection **Goal: Running code which modifies page content**

- Open `weather.js` and remove `getWeatherInfo("CA/San_Francisco", populateWeatherConditions);` from the document ready listener.
- In the document ready listener, we will populate a drop-down with some example cities:

```
$(function(){
    var cities = [
        { name: "London", code: "UK/London" },
        { name: "San Francisco", code: "CA/San_Francisco" },
        { name: "Cape Town", code: "ZA/Cape_Town" },
        { name: "Barcelona", code: "ES/Barcelona" },
        { name: "Boston", code: "NY/Boston" },
        { name: "New York", code: "NY/New_York" },
        { name: "Washington DC", code: "DC/Washington" },
        { name: "Tampa", code: "FL/Tampa" },
        { name: "Houston", code: "AL/Houston" },
        { name: "Montreal", code: "CYUL" },
        { name: "Los Angeles", code: "CA/Los_Angeles" },
        { name: "Miami", code: "FL/Miami" },
        { name: "West Palm Beach", code: "FL/West_Palm_Beach" }
    ];
    cities.forEach(function(city) {
        $("#city_menu").append("<option value='" + city.code + "'>" + city.name + "</option>");
    });
});
```

Clearing Displayed Data **Goal: Displaying new data when city selection changes**

When the user selects a new city the first thing we want to do is get rid of the old forecast content. The `emptyContent` function is simply clears the old weather information from the html page.

Note: If you decided to have a different layout this function will need to be specific to your custom display.

```
function emptyContent() {
    forge.logging.log("[emptyContent] removing old data");
    $("#forecast_information").empty();
    $("#current_conditions").empty();
    $("#forecast_conditions table tr").empty();

    forge.logging.log("[emptyContent] finished emptying content");
};
```

This function should be called every time new data is about to be displayed, which is handled by the `populateWeatherConditions`. Add the call to `emptyContent` at the top of the `populateWeatherConditions` function, which should then look like:

```
function populateWeatherConditions (weather) {
    var tmpl, output;

    emptyContent();

    forge.logging.log("[populateWeatherConditions] beginning populating weather conditions");

    tmpl = $("#forecast_information_tmpl").html();
    output = Mustache.to_html(tmpl, weather.current_observation);
    $("#forecast_information").append(output);
    forge.logging.log("[populateWeatherConditions] finished populating forecast information");

    tmpl = $("#current_conditions_tmpl").html();
    output = Mustache.to_html(tmpl, weather.current_observation);
    $("#current_conditions").append(output);
    forge.logging.log("[populateWeatherConditions] finished populating current conditions");

    tmpl = $("#forecast_conditions_tmpl").html();
    output = Mustache.to_html(tmpl, weather.forecast.simpleforecast);
    $("#forecast_conditions table tr").append(output);
    forge.logging.log("[populateWeatherConditions] finished populating forecast conditions");

    forge.logging.log("[populateWeatherConditions] finished populating weather conditions");
};
```

Remembering the previous location Goal: show different weather reports based on the selected city; and remember the previous selected city

The following code should be placed inside of the document ready listener.

When a city is selected from the drop-down list, we want to remember it to use it as the default city when the app is restarted.

To do that, we listen for changes to the `city_menu` element:

```
$("#city_menu").change(function() {
    var city = $("#city_menu option:selected").val();
    forge.prefs.set("city", city);
    getWeatherInfo(city, populateWeatherConditions);
});
```

See *forge.prefs.set*.

Using remembered locations Goal: default to the user's previously selected city when they re-open the app

When the application first runs we want to check if a city has already been saved from a previous run.

- the first time the app is run, this preference will be `null`, meaning its value has not been set
- if a city has been saved previously, it is selected in the drop-down list

```
forge.prefs.get("city", function(resource) {
    if (resource) { // user has previously selected a city
        var city = resource;
    } else { // no previous selection
        var city = "CA/San_Francisco";
    }
    $("#city_menu").val(city);
    $("#city_menu").change();
}, function (error) {
    forge.logging.error("failed when retrieving city preferences");
    $("#city_menu").val("CA/San_Francisco"); // default;
});
```

See *forge.prefs.get*.

The weather app should now be complete.

- Build and run the code

- Bask in all your glory, you have just written an app using Forge!

Reference app See the `part-4` tag in the Github repository for a reference app for this stage of the tutorial.

part-4.zip

What's next? It's easy to run the Weather App on a *different platform*.

Here are some *suggestions* on how to extend the weather app.

Weather App Conversion One of the key benefits of using Forge is being able to generate for multiple platforms from the same source code. Converting the Weather App from Chrome to mobile and vice-versa is as simple as making a few tweaks to the configuration file.

Chrome to Mobile If you did the Weather App tutorial using a Chrome extension you do not need to make any modifications at all. Simply go through the *Mobile getting started* section which will show you how to set up the development environment. Once you have the JDK, SDK and AVD configured simply follow the directions on how to *build* and *run* the app.

Mobile to Chrome The code used for building the Chrome extension will be identical, but we just need a way to display `index.html`.

One option is to have it show up as a popup that appears when a user clicks a toolbar button. To configure this, simply add the following button configuration to the `modules` section in `config.json` directory:

```
"button": {
    "default_popup": "index.html"
},
```

- the *default_popup* setting points to the HTML file that should be displayed when the toolbar button is clicked

If you want to know more about the configuration file, see the *Config File Reference*.

Now just *build* and *load* the extension to see the Weather Demo in Chrome.

Extensions and API This has been only a simple demo of what Forge is capable of. You can try extending the weather app with your own custom functionality. *Here* are a few suggestions to get you going. Also take a look at the *API documentation* to see what type of functionality is provided by Forge.

Possible Extensions If the tutorial has piqued your interest and you are looking for some more improvements to the weather app here are a few suggestions.

- The current UI is still quite basic. Why not add some CSS or change up the layout?
- Custom themes. Have a UI that allows the user to select a theme which is saved to preferences and persists after restarts
- Set the color of the temperature based on a numeric range
- The data returned by the weather API has loads more fields - customise the app for whatever your needs are

More challenging:

- Use geolocation to determine the current lat/lng

- Using reverse geocoding determine the current city (see http://trigger.io/cross-platform-application-development-blog/2012/05/15/how-to-build-a-location-based-hybrid-mobile-app-with-reverse-geocoding/)
- Use the information to look up the forecast for the current location

You could take a look at the *Forge API* documentation to help you implement this functionality or as a source of inspiration.

Tutorial: creating a browser add-on which interacts with pages

Tutorial Part 1 In this tutorial, we will use the Forge platform to create a browser extension which lets users attach text notes to profiles on Facebook.

As this particular app is based around modifying the contents of web pages on-the-fly, it is not applicable to mobile app development: see *our weather app tutorial* for how to get started with a mobile app.

Contents

Goal By the end of this part of the tutorial, you will have a development environment set up to embed your JavaScript code into Facebook pages.

What you'll need

- an internet connection
- Google Chrome: version 5 or higher
- a working knowledge of HTML, CSS and JavaScript debugging tools like the Chrome Developer Tools
- about an hour of your time

Preparation Firstly, follow the *Getting Started with Forge* instructions. This will leave you with a browser extension to be customised to your particular needs.

This will leave you with a `src` directory (where all your code will be placed), containing a `config.json` file.

Update the extension name and description A cosmetic change, but an important one, is setting the name and description for your extension. Depending on the browser, this may appear during the install process and on the list of installed extensions.

Open `config.json` from the `src` folder created above, and change the name and description:

```
"name": "Facebook note demo",
"description": "A demo application: lets users attach notes to Facebook profiles",
```

Building the app

Toolkit To build your Chrome extension using the Toolkit, simple click on the app you wish to build from the Your Apps screen, and then the 'Chrome' link. You will see the full traceback in the console as the commands are run so you can see progress and any warnings.

If you make subsequent code changes that you want to build and test on the same platform, just click 'Run again' at the bottom of the console view to rebuild it.

Command-line To build the extension, bring up your terminal, *find the location of the forge executable* for your platform, and enter the `forge build chrome` command.

```
$ forge build chrome
[   INFO] Forge tools running at version 1.0.1
[   INFO] configuration has changed: creating new templates
[   INFO] starting new build
[   INFO] build 11 started...
[   INFO] build completed successfully
[   INFO] current configuration hash is 564c9ef0ede72b76ce06b823047f075a
[   INFO] fetching new Forge templates
[   INFO] fetching unpackaged artefacts for build 532 into ".template"
[WARNING] creating output directory ".template"
[   INFO] Development build created. Use forge run to run your app.
```

Underneath the `development/chrome` directory, you now have a development Chrome extension which can be installed from the `chrome://extensions` screen. Following these instructions, create and load an extension.

Warning: Whenever you make changes to files in the `src` directory, you will need to rebuild the app with `forge build chrome`, then reload the extension by going to `chrome://extensions` and clicking on *Reload* in the relevant section.

Activating on the right pages This particular browser extension should only be activated on Facebook pages.

To set this up, we need to make sure the *activations* module is enabled and set the `patterns` array in `config.json`. To do this we need to add an `activations` object to `modules` in `config.json`, if it already exists you will need to edit it to look like this:

```
"modules": {
  "activations": [
    {
      "patterns": ["http://www.facebook.com/*", "https://www.facebook.com/*"],
      "styles": [],
      "scripts": []
    }
  ]
}
```

The "http://www.facebook.com/*" value is a match pattern dictating which URLs the extension should activate on.

Customise the JavaScript **Goal: run some JavaScript when Facebook pages load**

Currently, the extension doesn't run any JavaScript when your extension activates. In this section, we'll create a new JavaScript file and configure the extension to load it on the right pages.

Firstly, in the `src` directory, create a file called `fb-note-demo.js`. Open `fb-note-demo.js` in your preferred text editor, and add this code:

```
alert("Facebook demo extension loaded");
```

Include your JavaScript in your extension The JavaScripts we embed are defined in `src/config.json`, in the `scripts` array. Change this:

```
"scripts": []
```

to this:

```
"scripts": [
  "/fb-note-demo.js"
]
```

Now, *rebuild your extension and reload it in Chrome*. When you go to a Facebook page, you should see your own alert popup.

Reference extension fb-part-1.zip contains the code you should have at this point. Feel free to check your code against it, or use it to resume the tutorial from this point.

It's not working! Things to check:

- have you updated `src/config.json` to point at the modified local copy of `fb-note-demo.js`?
- have you reloaded your extension?
- on Facebook, use Chrome's Developer Tools to see which scripts have been embedded in the page: do you see a HTTP 404 for your JavaScript file?
- on Facebook, use the console in Chrome's Developer Tools to check for JavaScript errors: uncaught exceptions may cause the alert messages not to appear
- clearing your browser cache (at `chrome://history/#e=1&p=0`) will flush out any old resources
- still not working? Get in touch at support@trigger.io!

Tutorial Part 2 In the second part of this tutorial, we build on the previous steps to add the JavaScript code which lets users attach text notes to profiles on Facebook.

Contents

Goal By the end of this part of the tutorial, you will have a basic extension which lets users attach notes to Facebook profiles.

Identifying a Facebook profile **Goal: only show an alert when we’re on a Facebook profile page**

To show the correct note on each Facebook profile page, we will use a regular expression to pull the user ID out of a Facebook URL.

Facebook profile page URLs take two forms:

```
* ``www.facebook.com/profile.php?id=123456789``
* ``www.facebook.com/profile.name``
```

If we’re not on a page that has a URL like this, we don’t want to do anything.

Replace your alert with:

```
var oldProfile = new RegExp("\\.facebook\\.com/profile\\.php\\?id=(\\d+)");
var newProfile = new RegExp("\\.facebook\\.com/([A-Za-z0-9\\.]+)(\\?.+)?$");
var ignoreProfiles = new RegExp("\\.php$");

var oldProfileM = oldProfile.exec(document.URL),
  newProfileM = newProfile.exec(document.URL),
  personID = undefined;
if (oldProfileM) {
  // URL is like www.facebook.com/profile.php?id=123456789
  var personID = oldProfileM[1];
} else if (newProfileM) {
  // URL is like www.facebook.com/profile.name ...
  var personID = newProfileM[1];
  if (ignoreProfiles.exec(personID)) {
    // ... but it's a system page (like the login screen)
    var personID = undefined;
  }
}

if (typeof personID !== "undefined") {
  handleProfilePage(personID);
}

function handleProfilePage(personID) {
  alert("On a Facebook profile page");
}
```

We've used a regular expression which matches URLs like http://www.facebook.com/profile.php?id=123456789 to only continue with our code if we appear to be on a profile page. It also pulls out the unique user ID which we will soon use to attribute the right note to the right person - we save the unique ID into the `personID` variable.

Reading in existing notes Of course, we want to save notes away so that we can see them again next time we visit a profile page. To do that, we will use a feature of the Forge platform which lets us persist data between browser windows and across page loads and browser restarts; we call these data *preferences*.

To read a preference, we will use the *forge.prefs.get* method in `fb-note-demo.js`: add the following code inside the to the end of the `handleProfilePage` function:

```
forge.prefs.get("fb_notes", function(notes) {
  // set to null when no preference yet set
  var existingNotes = notes === null? {}: notes;

  if (existingNotes[personID] === undefined) {
    existingNotes[personID] = "";
  }
});
```

As *forge.prefs.get* is an asynchronous method, we have passed in a callback to process the stored value.

Note: The check for `existingNotes` being `null` is to handle the initial case where there are no persisted notes.

How do users change the note? Now we can read in existing notes, but how do we let users edit them?

Well, to do this we will create a small HTML snippet to be inserted into the Facebook page, which lets users edit the note. Add this code to the end of the `forge.prefs.get` callback:

```
var noteEl = document.createElement("div"),
  textarea = document.createElement("textarea");
noteEl.appendChild(textarea);
document.body.insertBefore(noteEl, document.body.firstChild);
```

Now, *rebuild your extension and reload it in Chrome*. On profile pages you should now see a `<textarea>` inserted right at the top of the page. We will cover how to *save* the note in a short while.

Using jQuery instead of plain JavaScript Interacting with the DOM can be a bit cumbersome in plain JavaScript: for this reason, you may want to include third-party libraries in your extension. To do so, we'll just change `src/config.json` to include jQuery in the list of scripts we include when the extension activates.

First, download http://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.min.js to your `src` directory, then update your `src/config.json`:

```
"scripts": ["jquery.min.js", "/fb-note-demo.js"]
```

So, we can replace the code from *the previous section* with this (adding a couple of element classes):

```
var noteEl = $("<div class='fb-note' style='z-index: 1000; position: absolute'>"+
  "<textarea class='fb-note-content'></textarea>"+
  "</div>");
$(".fb-note-content", noteEl).val(existingNotes[personID]);
$("body").first().prepend(noteEl);
```

Saving the note At this point, you should see a `textarea` at the top of Facebook profile pages. Next, we need to be able to save the contents of that textarea, so that the notes are persisted between page loads and browser restarts.

First, change the `noteEl` element to include a **save** link:

```
var noteEl = $("<div class='fb-note' style='z-index: 1000; position: absolute'>"+
  "<textarea class='fb-note-content'></textarea>"+
  "<a class='save-note' style='display:block'>save note</a>"+
  "</div>");
```

Then add a `click()` handler to that link:

```
$("a.save-note", noteEl).click(function() {
  // save textarea content into the existingNotes object we read before
  existingNotes[personID] = $(".fb-note-content", noteEl).val();
  // forge.prefs.set is the inverse of forge.prefs.get
  forge.prefs.set("fb_notes", existingNotes);
  alert("Note saved!");
});
```

Bringing it together Now, when you go to a Facebook profile page, you should see a `textarea` at the top of the page, along with a **save note** link. Editing the text and saving it should pop up an alert box - and the next time you visit that profile page, the saved message should be displayed.

You should be able to save different notes for different people.

Reference extension fb-part-2.zip contains the code you should have at this point. Feel free to check your code against it, or use it to resume the tutorial from here.

It's not working!

- You can use the standard Chrome JavaScript debugger to check for logic problems in your code
- JavaScript files can be cached, clear this out at `chrome://history/#e=1&p=0`
- are you seeing the wrong note for people? It's due to Facebook's strange page transitions: reloading the page should show the right note. In a real application, you could add a simple poller on `document.URL`
- if your JavaScript file has not been embedded in the page (i.e. you don't see it in the list of available scripts in the debugger), there's either a syntax error in your file (which will show up on the console) or there's a problem reading the file (which will show up as an 404 error in the *Resources* panel)

Common gotchas:

- remember to *rebuild your extension and reload it in Chrome* after changing any config or code
- can't figure out the problem? Get in touch at support@trigger.io!

Tutorial Part 3 In the final part of this tutorial, we build on Part II to improve the appearance and functionality of the notes interface.

Contents

Goal By the end of this part of the tutorial, your Facebook notes application will look much more appealing than before!

Making the note look nicer Rather than just a plain, unstyled textarea, let's make the note more appealing by having it look a bit like a Post-It note.

To do this, we'll need to embed a CSS file in the Facebook page to style our note. In `src/config.json`, change the `styles` array from:

```
"styles": [
],
```

to:

```
"styles": [
  "/fb-note-demo.css"
],
```

In the same folder as `fb-note-demo.js`, create `fb-note-demo.css` with the following content:

```
.fb-note  {
  background-color: #FEF49C;
  border: 1px solid #CEC46C;
  position: absolute;
  top: 10px;
  right: 10px;
  z-index:1000;
  -moz-box-shadow: 5px 5px 5px #555;
  -webkit-box-shadow: 5px 5px 5px #555;
  box-shadow: 5px 5px 5px #555;
}

.fb-note a.save-note {
  display: block;
  font-size: 140%;
  width: 280px;
  background-color: white;
  margin: 0px auto;
  text-align: center;
  margin-bottom: 3px;
  background-color: #EEEEFF;
}

.fb-note .fb-note-content {
  width: 280px;
  height: 170px;
  border: none;
  background-color: transparent;
}
```

Feel free to change this CSS as you see fit!

Reload a Facebook profile page, and your note should be a bit nicer to look at!

Adding a close button Now that the note is covering some of the page, we should really add a close button to let people temporarily hide it. Of course, in a real application, this setting should be persisted, and more complex show/hide functionality considered, but for this demo a simple approach is fine.

When building the note in your JavaScript, change `noteEl` to be:

```
var noteEl = $("<div class='fb-note' style='z-index: 1000; position: absolute'>"+
  "<a class='fb-close-note'>X</a>"+
  "<textarea class='fb-note-content'></textarea>"+
  "<a class='save-note'>save note</a>"+
  "</div>");
```

Then add a `click()` handler for the close button to the end of your `forge.prefs.get` callback:

```
$("a.fb-close-note", noteEl).click(function() {
  $(".fb-note").remove();
});
```

Finally, add some CSS styling for the close button in `fb-note-demo.css`:

```
.fb-note a.fb-close-note {
  float: right;
  color: red;
  font-weight: bold;
}
```

Reference extension fb-part-3.zip contains the code you should have at the end of this tutorial. Feel free to check your code against it, or build on it to improve the Facebook note demo add-on!

It's not working!

- if there's a problem with your CSS not being applied properly, check that you can access the stylesheet at the URL specified in `src/config.json`
- for JavaScript problems, setting a debugger breakpoint at the start of your *onLoad* function is often the best place to start.
- are you seeing the wrong note for people? It's due to Facebook's strange page transitions: reloading the page should show the right note. In a real application, you could add a simple poller on `document.URL`.

Note: `api.log(message)` is a useful function which outputs messages to the console; we find it much more convenient than having `alert()` calls everywhere!

Other useful functions This tutorial has not covered all of the API Forge exposes to your extensions: see the *API* documentation for a comprehensive list.

2.3.2 API

Questions? Ask them on StackOverflow or email us.

Features

Mobile - *see all...*

- *Address book*
- *Analytics*
- *Barcode / QR scanning*
- *Camera*
- *Child browser*
- *Cross-domain ajax*
- *Custom URL schemes*
- *Events*
- *Facebook SDK access*
- *Foreground camera*
- *Files*
- *Geolocation*
- *In-app payments*
- *Media*
- *Native UI enhancements*
- *Offline*
- *Push notifications*
- *Reload*
- *SMS*
- *Storage*
- *Topbar native UI*
- *Tabbar native UI*

Native plugins - *learn more...*

Browser add-ons - *see all...*

- *Background page*
- *Content scripts*
- *Cross-domain ajax*
- *Messaging*
- *Storage*
- *Toolbar button*

Mobile

`barcode`: Barcode / QR Code scanner The `forge.barcode` namespace allows the user to scan a barcode using the devices camera and returns its content

Config The `barcode` module must be enabled in `config.json`

```
{
    "modules": {
        "barcode": true
    }
}
```

API

`scan` Platforms: iOS, Android

Show a UI that allows the user to scan a barcode and return its value

Example:

```
forge.barcode.scan(function (value) {
    alert("You scanned: "+value);
});
```

`barcode.`**`scan`**(*success*, *error*)

Arguments

- **success** (*function(value)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`scanWithFormat` Platforms: iOS, Android

Show a UI that allows the user to scan a barcode and return its value and type.

Example:

```
forge.barcode.scanWithFormat(function (barcode) {
  alert('You scanned a '+barcode.format+': '+barcode.value);
});
```

`barcode.`**`scanWithFormat`**(*success*, *error*)

Arguments

- **success** (*function(barcode)*) – callback to be invoked when no errors occur - `barcode` will contain `format` and `value` keys, where `format` is the barcode type as returned by ZXing
- **error** (*function(content)*) – called with details of any error which may occur

`contact`: Accessing contacts Platforms: Mobile

The `forge.contact` namespace allows access to the native contact address book on the device the app is running on.

Contacts are represented by a simple Javascript object which follows the W3C Contacts API as much as possible.

Config The `contact` module must be enabled in `config.json`

```
{
    "modules": {
        "contact": true
    }
}
```

API

`select` Platforms: Mobile

Prompts the user to select a contact and returns a contact object.

`contact.select`(*success*, *error*)

Arguments

- **success** (*function(contact)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`selectAll` Platforms: Mobile

Returns a list of all the available contact IDs and contact name.

Due to performance limitations on Android devices, this method is unable to return the full list of fully populated contact names; our recommended pattern is to use this method to get the list of all available contact IDs, then lazily load the more detailed full contact information with the `selectById` method.

`contact.selectAll`(*success*, *error*)

Arguments

- **success** (*function(contactList)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`selectById` Platforms: Mobile

Returns more detailed information about a contact whose contact ID we already know.

`contact.selectById`(*id*, *success*, *error*)

Arguments

- **success** (*function(contact)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

Contact object When using `selectAll`, the returned contacts list would look something like:

```
[
    {
        ":ref:`id <field-contact-id>`": "14894",
        ":ref:`displayName <field-contact-displayName>`": "Mr Joe Bloggs"
    },
    {
        ":ref:`id <field-contact-id>`": "481516",
        ":ref:`displayName <field-contact-displayName>`": "Mr John Locke"
    },
    ...
]
```

Below is an example of a contact object returned from `select` or `selectById`, with details for some field types.

```
{
  "id": "14894",
  "displayName": "Mr Joe Bloggs",
  "name": {
    "formatted": "Mr Joe Bloggs",
    "familyName": "Bloggs",
    "givenName": "Joe",
    "middleName": null,
    "honorificPrefix": "Mr",
    "honorificSuffic": null
  },
  "nickname": "Joe",
  "phoneNumbers": [
    {
      "value": "+447554639203",
      "type": "work",
      "pref": false
    }
  ],
  "emails": [
    {
      "value": "joe-bloggs@trigger.io",
      "type": "work",
      "pref": false
    }
  ],
  "addresses": [
    {
      "country": "United Kingdom",
      "formatted": "1-11 Baches Street\nLondon\nLondon\N1 6DL\nUnited Kingdom",
      "locality": "London",
      "postalCode": "N1 6DL",
      "pref": false,
      "region": "London",
      "streetAddress": "1-11 Baches Street",
      "type": "work"
    }
  ],
  "ims": [
    {
      "value": "joe-bloggs@trigger.io",
      "type": "gtalk",
      "pref": false
    }
  ],
  "organizations": [
    {
      "department": "Product development",
      "name": "Forger",
      "pref": false,
      "title": "Software engineer",
      "type": null
    }
  ],
  "birthday": "1983-11-23",
  "note": "Any text can go here",
  "photos": [
    {
      "value": "data:image/jpg;base64,ABCDEF1234",
      "type": null,
      "pref": false
    }
  ],
  "categories": null,
  "urls": [
    {
      "value": "http://trigger.io",
      "type": "homepage",
      "pref": false
    }
  ],
}
```

Fields This section includes more detailed information on the contents of fields with non-obvious content.

id This is a unique identifier for the contact, and is guaranteed to be the same if the user selects the same contact again.

displayName This is a formatted version of the contacts name which can be used for display. On iOS this is generated from the various parts of the name, on Android this is stored as a separate value.

name This is an object containing the various parts of the contacts name, including a formatted version which is used as the previous displayName value.

nickname A string value containing a nickname for the contact

phoneNumbers An array of objects containing details of a contacts phone numbers. Each number has a `value`, a `type` (such as `home` or `work`) and also a `pref` property, which is unsupport on Android and iOS so is always false.

emails Similarly this property is an array of objects describing a contacts emails, with `value`, `type` and `pref` (which is also always false).

addresses An array of objects describing a contacts addresses, `formatted` contains a string generated from the other properties which can be used to display the address. Each object also contains a `pref` property which is always false.

ims Contains an array of Instant Messaging details for a contact, formatted similarly to phoneNumbers and emails.

organizations Contains an array of objects describing organizations the contact is part of.

Can only contain one organization on iOS.

birthday Contains a string with the date of birth of the contact.

note A string which can contain arbitrary information about the contact.

photos Contains an array of thumbnail photos associated with the contact, each photo has a value which contains a `data:` uri of the image. The `type` and `pref` properties are not used.

Contains at most 1 photo on iOS.

categories Not available on iOS or Android.

urls Contains an array of URLs related to the contact, formatted similarly to phoneNumbers and emails.

Permissions On Android this module will add the `READ_CONTACTS` permission to your app, users will be prompted to accept this when they install your app.

`display`: App display options This controls how your app will be displayed as the device is moved around. The default is to allow for any orientation, with the content being re-drawn as the screen is rotated.

Config

```
{
    "modules": {
        "display": {
            "orientations": {
                "default": "any",
                "iphone": "portrait",
                "ipad": "landscape",
                "android": "landscape",
            },
            "fullscreen": {
                "android": false,
                "ios": false,
                "wp": true
            }
        }
    }
}
```

You can limit this behaviour by specifying the desired supported orientations as `orientations.default`, choosing from `"any"`, `"portrait"` or `"landscape"`. You can further customise this behaviour by specifying orientation support for different devices, e.g. `orientations.iphone` and `orientations.ipad`. For example:

```
"orientations": {
  "default": "any",
  "iphone": "portrait",
  "ipad": "landscape"
},
```

This configuration means

- by default, display your app in any orientation
- ... but on iPhones, only display your app in portrait mode, either way up
- ... and on iPads, your app will only use the landscape orientation
- ... on Android the default will apply and any orientation allowed by the device will be used

The fullscreen option will hide the system statusbar while your app is running and allow your app to run completely fullscreen on the device. Supported device types are:

- `android`: Android devices
- `ios`: Both iPhone and iPad devices
- `wp`: Windows phone devices

API The `display` module also allows you to change the orientation limitations while your app is running with the following API.

`orientation.forceLandscape` **Platforms: Mobile**

Force the app into a landscape orientation.

`display.orientation.`**`forceLandscape`**`(`*success*, *error*`)`

Arguments

- **success** (*function(value)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`orientation.forcePortrait` **Platforms: Mobile**

Force the app into a portrait orientation.

`display.orientation.`**`forcePortrait`**`(`*success*, *error*`)`

Arguments

- **success** (*function(value)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`orientation.allowAny` **Platforms: Mobile**

Allow any app orientation.

`display.orientation.`**`allowAny`**`(`*success*, *error*`)`

Arguments

- **success** (*function(value)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`document:` Document utility functions Internet Explorer features a number of inconsistencies with some common document operations, these methods allow you to work around these problems in a platform-independent manner.

API

`reload` **Platforms: Browser Only**

`document.`**`reload`**`()`

Internet Explorer's implementation of document.reload() only refreshes the static content of a page. This method will reload a page and force all scripts to be re-evaluated as well.

`location` Platforms: All

`document.location(success, error)`

Arguments

- **success** (*function(location)*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

Internet Explorer versions 9 and later aborts page loading with a permission denied message if the document location is accessed from within an iframe. This function provides a workaround which functions correctly under all browsers.

Error object properties:

- `statusCode`: Status code returned from the server.
- `content`: Content returned from the server (if available).

Example:

```
forge.document.location(function(location) {
    forge.logging.log(location.href);
  }, function(error) {
    alert('Failed to get document location: '+error.message);
  }
});
```

`event`: Events

Config

The `event` module must be enabled in `config.json`

```
{
    "modules": {
        "event": true
    }
}
```

API

The `forge.event` namespace allows app to listen for events of interest, which may be triggered multiple times (or potentially not at all) depending on the situation.

`menuPressed.addListener` Platforms: Android

Triggered when the menu button is pressed on an Android device.

`event.menuPressed.addListener(callback, error)`

Arguments

- **callback** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`backPressed.addListener` Platforms: Android

Triggered when the back button is pressed on an Android device.

`event.backPressed.addListener(callback, error)`

Arguments

- **callback** (*function(closeApplication)*) – callback invoked when back button is pressed, the first argument is a function which if called will close the application.
- **error** (*function(content)*) – called with details of any error which may occur

`backPressed.preventDefault` Platforms: Android

Prevents the default action when the back button is pressed from the point this is called onwards, allowing the app to handle the event itself using `backPressed.addListener`.

`event.backPressed.`**`preventDefault`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`orientationChange.addListener` Platforms: Mobile

Triggered when the device is rotated, use forge.is.orientation.portrait() and forge.is.orientation.landscape() to determine orientation.

`event.orientationChange.`**`addListener`**`(`*callback*, *error*`)`

Arguments

- **callback** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`connectionStateChange.addListener` Platforms: Mobile

Triggered when the device connection state changes, use forge.is.connection.connected() and forge.is.connection.wifi() to test new connection state.

This event will also be fired once during app startup, as soon as we determine the connection status.

`event.connectionStateChange.`**`addListener`**`(`*callback*, *error*`)`

Arguments

- **callback** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`messagePushed.addListener` Platforms: Mobile

Triggered when a push notification is received both while the application is running or if the application is launched via that notification.

Currently available as part of our *integration with Parse*.

`event.messagePushed.`**`addListener`**`(`*callback*, *error*`)`

Arguments

- **callback** (*function(data)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`appPaused.addListener` Platforms: Mobile

Triggered when the app loses focus and moves into the background. At this point what can be executed varies by platform:

- Android: Any javascript can be run, but timers may not be fired until the app is resumed, this prevents unnecessary battery usage by the app.
- iOS: A short amount of time is given for execution, it is generally best to assume that callbacks and timers may not fire until the app is resumed.
- Windows Phone: Any javascript will be executed only when the app resumes.

You should also not assume that an app that is paused will be resumed, the app may be killed at this point by the user or device without ever being resumed.

`event.appPaused.`**`addListener`**`(`*callback*, *error*`)`

Arguments

- **callback** (*function(data)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`appResumed.addListener` Platforms: Mobile

Triggered when the app is resumed from a previous paused state.

`event.appResumed.`**`addListener`**`(`*callback*, *error*`)`

Arguments

- **callback** (*function(data)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`facebook`: Facebook SDK access The `forge.facebook` namespace allows access to the native Facebook SDK, which provides similar functionality to the JavaScript SDK with the additional feature of supporting SSO (Single Sign-On) between the users Facebook app and your Forge app.

You can see a demo app that makes use of the Facebook module in this screencast, with the code on Github.

Important: Using the Facebook module requires you to have a Facebook account and signup to Facebook's developer platform. Your relationship with Facebook is separate from your relationship with Trigger.io. If you include the Facebook module in your app, the App Id will report information about app installs and usage to Facebook with that information also made available on your dashboard at https://developers.facebook.com. You must disclose to your users that your app passes the App Id to you and Facebook.

Config The `facebook` module must be enabled in `config.json`

```
{
    "modules": {
        "facebook": {
            "appid": "123456789012345"
        }
    }
}
```

Note: To use this module you will need to setup your app in Facebook. More information can be found in the Tips section below and at https://developers.facebook.com/

API

`facebook.authorize` Platforms: Mobile

Authorize the current user with Facebook, may show a login UI if new permissions are required, or a valid login token is not available (i.e. on first login).

The `audience` parameter is only used on iOS (due to differences in the Facebook SDK), but can be passed in on Android to no ill effect. It should be one of:

- "everyone" - by default, your app's updates are public
- "friends" - by default, the user's friends can see your your app's updates
- "only_me" - by default, just the user can see the updates
- "none" - no one can see the updates by default

The success callback will be called with information about the users access_token, you can store and use this token following the Facebook developer guidelines.

`facebook.`**`authorize`**(*[permissions]*, *success*, *error*)

Arguments

- **permissions** (*array*) – An optional array of permissions to request
- **audience** (*array*) – Optional string indicating who should see updates by default
- **success** (*function(token_information)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`facebook.hasAuthorized` Platforms: Mobile

Takes the same options and returns the same data as `facebook.authorize`, but will not prompt the user for login if required. Used to only log in a user if their interaction is not required.

For more information about the `audience` parameter, see the *facebook.authorize*.

`facebook.`**`hasAuthorized`**(*[permissions]*, *success*, *error*)

Arguments

- **permissions** (*array*) – An optional array of permissions to request
- **audience** (*array*) – Optional string indicating who should see updates by default
- **success** (*function(token_information)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`facebook.logout` Platforms: Mobile

Logout the current user and clear any cached login details.

`facebook.logout(`*success*, *error*`)`

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`facebook.api` Platforms: Mobile

Make a Facebook Graph API call. See https://developers.facebook.com/docs/reference/javascript/FB.api/ for further details.

`facebook.api(`*path*[[, *method*], *params*], *success*, *error*`)`

Arguments

- **path** (*string*) – API path to call, i.e. `"me/posts"`
- **method** (*string*) – Type of request, i.e. `"GET"`
- **params** (*object*) – Additional parameters for the request, i.e. `{limit: 5}`
- **success** (*function(response)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`facebook.ui` Platforms: Mobile

Display a Facebook dialog UI. See https://developers.facebook.com/docs/reference/javascript/FB.ui/ for further details.

Note that if the user hits "Cancel" in the dialog, your success callback will still be called - with `{}` as its parameter. This is the behaviour of the underlying Facebook SDK - for more information, see http://stackoverflow.com/a/13729707/29903.

`facebook.ui(`*params*, *success*, *error*`)`

Arguments

- **params** (*object*) – Dictionary of paramters, must include `method`
- **success** (*function(response)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

Tips For a quick tutorial on setting up your app in Facebook to enable login and open graph API calls see our demo app build instructions

General

- To use the Facebook module a Facebook app needs to be created on https://developers.facebook.com/apps. Additionally, on the app configuration page, "Native iOS App" and "Native Android App" need to be enabled, and within each of those sections SSO should also be enabled.
- If a user revokes your apps access, or logs out from the Facebook app you may get OAuth errors returned from API calls, in this situation you should call `forge.facebook.logout()` and reauthorize the user.

Android

- On Android a hash of the key used to sign your app is required by Facebook to confirm your app should be allowed to access the Facebook API. The easiest way to configure this is to simply start using the Facebook API, any API methods will return an error message which includes the hash and the URL to visit to configure it.

iOS

- On iOS you must add your applications bundle id to the Facebook developer app settings page. You set a specific bundle id using the package_names module.

`file`: File and Camera access The file API allows storage of files on the local system as well as capturing images with the camera or selecting them from the users saved photos.

Notes

- File objects are simple Javascript objects which contain at least a `uri`. They can be serialised using JSON.stringify and safely stored in Forge preferences.
- The `uri` parameter can be used directly on some platforms. This is not recommended - instead use the provided helper function `forge.file.URL`.
- Image orientation is automatically handled where possible: if a camera photo contains rotation information it will be correctly rotated before it is displayed or uploaded.
- Files can be uploaded by including them as an array in `request.ajax()`. For example if `myFile1` and `myFile2` were images returned by `file.getImage()`:

```
forge.request.ajax({
    url: "http://example.com/file_upload",
    files: [myFile1, myFile2]
});
```

Note: For more information about how to cache remote files in your app, see *Caching files*.

Config The `file` module must be enabled in `config.json`

```
{
    "modules": {
        "file": true
    }
}
```

API

`getImage` **Platforms: Mobile**

Returns a file object for a image selected by the user from their photo gallery or (if possible on the device) taken using their camera.

The optional parameters can contain any combination of the following:

- `width`: The maximum height of the image when used, if the returned image is larger than this it will be automatically resized before display. The stored image will not be resized.

- `height`: As `width` but sets a maximum height, both `height` and `width` can be set.
- `source`: By default the user will be prompted to use the camera or select an image from the photo gallery, if you want to limit this choice you can set this to `"camera"` or `"gallery"`.
- `saveLocation`: By default camera photos will be saved to the device photo album, with this setting they can be forced to be saved within your application by using `"file"`.

Returned files will be accessible to the app as long as they exist on the device.

`file.`**`getImage`**([*params*], *success*, *error*)

Arguments

- **params** (*object*) – object optional parameters.
- **success** (*function(file)*) – callback to be invoked when no errors occur (argument is the returned file)
- **error** (*function(content)*) – called with details of any error which may occur

Important: On iOS devices, the first time your app reads from the gallery, the user will be prompted to allow the app to access your location. This is because the EXIF data in images stored there could be used to infer a user's geolocation. For more information, see *Permissions*.

`getVideo` Platforms: Mobile

Returns a file object for a video selected by the user from their photo gallery or (if possible on the device) taken using their camera.

The optional parameters can contain any combination of the following:

- `source`: By default the user will be prompted to use the camera or select a video from the photo gallery, if you want to limit this choice you can set this to `"camera"` or `"gallery"`.

Returned files will be accessible to the app as long as they exist on the device.

`file.`**`getVideo`**([*params*], *success*, *error*)

Arguments

- **params** (*object*) – object optional parameters.
- **success** (*function(file)*) – callback to be invoked when no errors occur (argument is the returned file)
- **error** (*function(content)*) – called with details of any error which may occur

Important: On iOS devices, the first time your app reads from the gallery, the user will be prompted to allow the app to access your location. This is because the EXIF data in files stored there could be used to infer a user's geolocation. For more information, see *Permissions*.

`getLocal` Platforms: Mobile

Returns a file object for a file included in the `src` folder of your app

`file.`**`getLocal`**(*path*, *success*, *error*)

Arguments

- **path** (*string*) – Path to the file, i.e. `"images/home.png"`.

- **success** (*function(file)*) – callback to be invoked when no errors occur (argument is the returned file)
- **error** (*function(content)*) – called with details of any error which may occur

`cacheURL` Platforms: Mobile

Downloads a file at a specified URL and returns a file object which can be used for later access. Useful for caching remote resources such as images which can then be accessed directly from the local filesystem at a later date.

Cached files may be removed at any time by the operating system, and it is highly recommended you use the `isFile` method to check a cached file is still available before using it.

`file.`**`cacheURL`**(*url*, *success*, *error*)

Arguments

- **url** (*string*) – URL of file to cache.
- **success** (*function(file)*) – callback to be invoked when no errors occur (argument is the returned file)
- **error** (*function(content)*) – called with details of any error which may occur

`saveURL` Platforms: Mobile

Downloads a file at a specified URL and returns a file object which can be used for later access. Saves the file in a permanant location rather than in a cache location as with `cacheURL`.

Important: Files downloaded via this method will not be removed if you do not remove them, if the file is only going to be used temporarily then `cacheURL` is more appropriate.

`file.`**`saveURL`**(*url*, *success*, *error*)

Arguments

- **url** (*string*) – URL of file to save.
- **success** (*function(file)*) – callback to be invoked when no errors occur (argument is the returned file)
- **error** (*function(content)*) – called with details of any error which may occur

`isFile` Platforms: Mobile

Returns true or false based on whether a given object is a file object and points to an existing file on the current device.

`file.`**`isFile`**(*file*, *success*, *error*)

Arguments

- **file** (*file*) – the file object to check
- **success** (*function(isFile)*) – callback to be invoked when no errors occur (argument is a boolean value).
- **error** (*function(content)*) – called with details of any error which may occur

`URL` **Platforms: Mobile**

Returns a URL which can be used to display an image. Height and width will be limited by the values given when originally selecting the image.

`file.`**`URL`** (*file*, *success*, *error*)

Arguments

- **file** (*file*) – the file object to load data from
- **success** (*function(url)*) – callback to be invoked when no errors occur, first argument is the image URL
- **error** (*function(content)*) – called with details of any error which may occur

`base64` **Platforms: Mobile**

Returns the base64 value for a files content.

`file.`**`base64`** (*file*, *success*, *error*)

Arguments

- **file** (*file*) – the file object to load data from
- **success** (*function(base64String)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`string` **Platforms: Mobile**

Returns the string value for a files content.

`file.`**`string`** (*file*, *success*, *error*)

Arguments

- **file** (*file*) – the file object to load data from
- **success** (*function(string)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`remove` **Platforms: Mobile**

Delete a file from the local filesystem, will work for cached files but not images stored in the users photo gallery.

`file.`**`remove`** (*file*, *success*, *error*)

Arguments

- **file** (*file*) – the file object to delete
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`clearCache` Platforms: Mobile

Deletes all files currently saved in the local cache.

`file.clearCache(success, error)`

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

Permissions On Android this module will add the `WRITE_EXTERNAL_STORAGE` permission to your app, users will be prompted to accept this when they install your app.

On iOS, accessing files in the device's gallery causes the user to be prompted to give your app access to their location. This is because files in the gallery may contain EXIF data, including geolocation and timestamps.

To avoid the user being shown this prompt, you could save your image into a file rather than the gallery, using the `saveLocation` parameter. This is not yet supported when capturing videos.

If a user chooses not to share their location with your app, the error callback of the method trying to read files from the gallery will be invoked.

`flurry`: Analytics with Flurry The `forge.flurry` APIs allow you to access the native Flurry SDK, which provides usage tracking and analytics via Flurry.

Before you can use this module, you will need to have set up a Flurry application, and know its API key.

Config The `flurry` module must be enabled in `config.json` with API keys specified for Android and/or iOS.

```
{
    "modules": {
        "flurry": {
            "android_api_key": "VMGKY81MY88PCDR38ARM",
            "ios_api_key": "3RAG3ZW4QXMRVNJH092S"
        }
    }
}
```

By just including this configuration in your app config, basic app analytics information - such as sessions, active users and new users - will be available in your Flurry dashboard.

For more advanced analytics, you can use the API methods described below.

API

`flurry.customEvent` Platforms: Mobile

Send a named and optionally parameterised event to Flurry. You could use this to track a user's navigation through your app, for example.

`flurry.customEvent(name[, parameters], success, error)`

Arguments

- **name** (*string*) – a name to identify this event

- **parameters** (*object*) – an optional hash of extra information that will be stored with this event
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`flurry.startTimedEvent` Platforms: Mobile

Takes the same options as `flurry.customEvent`, but using `flurry.endTimedEvent` you are able to easily measure the time it takes for your users to move from one action to another in your app.

`flurry.`**`startTimedEvent`**(*name*[, *parameters*], *success*, *error*)

Arguments

- **name** (*string*) – a name to identify this event
- **parameters** (*object*) – an optional hash of extra information that will be stored with this event
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`flurry.endTimedEvent` Platforms: Mobile

Mark the end of a particular timed event: the `name` parameter should match the `name` parameter passed into `flurry.startTimedEvent`.

`flurry.`**`endTimedEvent`**(*name*, *success*, *error*)

Arguments

- **name** (*string*) – a name to identify this event: should match the name passed into `flurry.startTimedEvent`
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`flurry.setDemographics` Platforms: Mobile

Store some demographic data about the current user, to enable more advanced filtering and grouping in the Flurry dashboard.

The `demographics` object should contain some or all of these keys:

- `user_id`: (string) a unique ID for the current user
- `age`: (number) the current user's age
- `gender`: (string) either "m" or "f"

`flurry.`**`setDemographics`**(*demographics*, *success*, *error*)

Arguments

- **demographics** (*string*) – a hash optionally including values for `user_id`, `age` and `gender`
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`flurry.setLocation` Platforms: Mobile

Set the user's current location: we recommend you use the *geolocation* module to grab the coords object which should be passed in. E.g.:

```
forge.geolocation.getCurrentPosition( function (position) {
    forge.flurry.setLocation(position.coords);
});
```

`flurry.`**`setLocation`**`(`*coords*, *success*, *error*`)`

Arguments

- **coords** (*object*) – hash representing location - must include `latitude`, `longitude` and `accuracy`.
- **success** (*function(response)*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`geolocation:` Geolocation Although geolocation APIs are part of the HTML5 specification, on some platforms, the default permissions dialogs can be cumbersome and annoying to your users.

For that reason, we offer an alternative way to get geolocation data.

Config The `geolocation` module must be enabled in `config.json`

```
{
    "modules": {
        "geolocation": true
    }
}
```

API

`geolocation.`**`getCurrentPosition`**`(`*options*, *success*, *error*`)`

Arguments

- **options** (*object*) – request specific levels of service from the location provider, currently supports `"enableHighAccuracy": true` to request GPS location if available.
- **success** (*function(position)*) – called with an object matching the W3C Position specification
- **error** (*function(error)*) – called when the user chooses not to share their location with your app

Note: To enable easy porting from existing HTML5 code onto Forge, we also accept parameters in the order `(success, error, options)`

Permissions On Android this module will add the `ACCESS_FINE_LOCATION` permission to your app, users will be prompted to accept this when they install your app.

`icons:` App icons This part of the config allows you to define the icons to be used for your app. All icons are square, and must be placed in your `src` directory.

Define your desired icons with `"size": "path"` attributes, where `size` is the pixel height (and width) of the icon, and `path` is where the image has been placed under the `src` directory.

You can specify different icons for different platforms as so:

```
"android": {
    "36": "icon36.png",
    "48": "icon48-android.png",
    "72": "icon72.png"
},
"chrome": {
    "16": "icon16.png",
    "48": "icon48-chrome.png",
    "128": "icon128.png"
}
```

Here, Android and Chrome will share their 16x16 pixel icon, but use different 48x48 pixel icons.

The icons required for each platform are listed below:

- Android: 36px, 48px and 72px
- Chrome: 16px, 48px and 128px
- Firefox: 32px and 64px
- Internet Explorer: 16px `.ico` format
- iOS: 57px, 72px, 114px and 144px for home screen icons, 512px to be shown in iTunes.
- Safari: 32px, 48px and 64px with transparent background. See the Creating an Image section in Apple's Safari extension guide.

Note: If you specify *any* icons for a particular platform, you **must** specify all required icons!

Note: iOS includes a special `prerendered` option, setting this to true will stop iOS from applying the gloss effect to your icons.

Config

```
{
    "modules": {
        "icons": {
            "android": {
                "36": "icon36.png",
                "48": "icon48-android.png",
                "72": "icon72.png"
            },
            "ios": {
                "57": "icon57.png",
                "72": "icon72-ios.png",
                "114": "icon114.png",
                "144": "icon144.png",
                "prerendered": true
            }
        }
    }
}
```

is: Platform Detection Forge allows you to build cross-platform mobile apps and browser extensions from the same code. Sometimes it may be necessary to do a specific action based on the platform that is running the code. The following methods allow you to determine what platform that is.

Config The `is` module must be enabled in `config.json`

```
{
    "modules": {
        "is": true
    }
}
```

API

API

mobile
`is.mobile()`

Return boolean Returns true if running on a mobile device

desktop
`is.desktop()`

Return boolean Returns true if running on a desktop/laptop computer

web
`is.web()`

Return boolean Returns true if running on as a hosted web app

android
`is.android()`

Return boolean Returns true if running on an Android device

ios
`is.ios()`

Return boolean Returns true if running on an IOS device

chrome
`is.chrome()`

Return boolean Returns true if running on Chrome browser

firefox
`is.firefox()`

Return boolean Returns true if running on Firefox browser

safari
`is.safari()`

Return boolean Returns true if running on Safari browser

ie
`is.ie()`

Return boolean Returns true if running on IE browser

orientation **Platforms: Mobile**

portrait
`is.orientation.portrait()`

Return boolean Returns true if a mobile device has a portrait orientation

landscape
`is.orientation.landscape()`

Return boolean Returns true if a mobile device has a landscape orientation

connection **Platforms: Mobile**

Note: These functions are not reliable during your app's initialisation: you should use *connectionStateChange.addListener*. We guarantee to fire that event as the app starts up.

---

connected
`is.connection.connected()`

Return boolean Returns true if a mobile device has an active internet connection.

wifi
`is.connection.wifi()`

Return boolean Returns true if a mobile device is connected via wifi.

launchimage: Launch images The `launchimage` module displays an image while your app is loading.

By default, the launch image is hidden automatically when the window `load` event fires - this gives your JavaScript some time to initialise itself, set up native *topbars* and *tabbars* for example.

However, if you want full control over hiding the launch image yourself, set `hide-manually` to `true` (see below).

Config Images to be displayed during launch as required on iOS, for further details see the Apple documentation.

On Android the image will be displayed centered while the first page is loading: as Android device sizes vary a pixel perfect loading image cannot be used. The image will be proportionally scaled down to fit the screen if necessary. The solid color to use behind this launch image can be configured with the `background-color` configuration directive seen below.

All 7 iOS images must be defined for any to be included in iOS builds. Both Android images must be defined for Android builds.

Properties and image sizes are:

- `iphone`: 320x480px
- `iphone-retina`: 640x960px
- `iphone-retina4`: 640x1136px - for the iPhone 5's four inch screen
- `ipad`: 768x1004px
- `ipad-landscape`: 1024x748px
- `ipad-retina`: 1536x2008px
- `ipad-landscape-retina`: 2048x1496px
- `android`
- `android-landscape`

```
{
    "modules": {
        "launchimage": {
            "iphone": "iphone.png",
            "iphone-retina": "iphone-retina.png",
            "iphone-retina4": "iphone-retina4.png",
            "ipad": "ipad.png",
            "ipad-landscape": "ipad-landscape.png",
            "ipad-retina": "ipad-retina.png",
            "ipad-landscape-retina": "ipad-landscape-retina.png",
            "android": "android.png",
            "android-landscape": "android-landscape.png",
            "hide-manually": true,
            "background-color": "#000000"
        }
    }
}
```

API

`hide` Platforms: Mobile

`launchimage.hide`(*success*, *error*)

Arguments

- **success** (*function(content)*) – called after the launch image has been hidden
- **error** (*function(content)*) – called with details of any error which may occur

If you want to manually hide the launch image yourself, use this API.

You will probably also want to specify `hide-manually` in your launchimage module configuration.

`logging`: Logging Allows you to log a message, and optionally an exception, to the console service provided by the underlying platform.

Config The `logging.level` configuration directive controls the verbosity of the logging system. It should be set to one of `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`. A setting of `DEBUG` means that all messages will be logged, whereas a setting of `CRITICAL` means that only messages of level `CRITICAL` will be logged.

```
{
    "modules": {
        "logging": {
            "level": ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]
        },
    }
}
```

Platforms: Internet Explorer only

The `logging.console` configuration directive enables a log window for the add-on's background page.

```
"logging": {
    "console": true
},
```

API

`log` Platforms: All

`logging.`**`log`**`(`*message[, exception][, level]*`)`

Arguments

- **message** – a string to log on the browser console
- **exception** – a JavaScript Error instance: details from the exception will be appended to your log message
- **level** – importance of this message: one of `forge.logging.DEBUG`, `forge.logging.INFO`, `forge.logging.WARNING`, `forge.logging.ERROR` or `forge.logging.CRITICAL`

`logging.`**`debug`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.DEBUG)`

`logging.`**`info`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.INFO)`

`logging.`**`warning`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.WARNING)`

`logging.`**`error`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.ERROR)`

`logging.`**`critical`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.CRITICAL)`

`media`: Media file playback

Config The `media` module must be enabled in `config.json`

```
{
    "modules": {
        "media": {
            "enable_background_audio": true
        }
    }
}
```

- `enable_background_audio`: If this is true then audio players will continue to play even when the app is running in the background.

API

`videoPlay` Platforms: Mobile

Play a video at a URL fullscreen on the device.

Note: Allow users to select or capture videos using their device you may use the *file module*.

`media.videoPlay` (*url*, *success*, *error*)

Arguments

- **url** (*string*) – URL to video.
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`createAudioPlayer` Platforms: Mobile

Create a audio player object from a media file which can then be used to play the audio.

The audio player object returned in the success callback has the following methods, all of which take success and error callbacks in the same manner as other Forge methods:

- `player.play(success, error)`: Begin (or resume) playing the audio file.
- `player.pause(success, error)`: Pause the playback of the file.
- `player.stop(success, error)`: Stop the playback of the file and release the audio system.
- `player.duration(success, error)`: Calls the success callback with the duration of the audio in seconds.
- `player.seek(seekTo, success, error)`: Seek to the given time (in seconds) in the audio file, if the file is playing it will continue to play after seeking.

Warning: For `player.duration` and `player.seek` to work properly on Android, use 128kb/s constant bit-rate MP3 files.

Example:

```
forge.file.getLocal("music.mp3", function (file) {
  forge.media.createAudioPlayer(file, function (player) {
    player.play();
  });
});
```

`media.`**`createAudioPlayer`**(*file*, *success*, *error*)

Arguments

- **file** (*string*) – File object created using forge.file methods representing audio object.
- **success** (*function(player)*) – callback to be invoked when player has been created
- **error** (*function(content)*) – called with details of any error which may occur

`notification`: Notifications

Config The `notification` module must be enabled in `config.json`

```
{
    "modules": {
        "notification": true
    }
}
```

API Notifications allow you to send alerts.

`create` Platforms: All

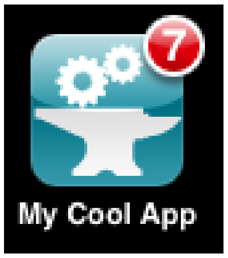
`notification.`**`create`**(*title*, *text*, *success*, *error*)

Arguments

- **title** (*string*) – title
- **text** (*string*) – notification message
- **success** (*function()*) – success
- **error** (*function(content)*) – called with details of any error which may occur

`setBadgeNumber` Platforms: iOS

Allows you to set or remove a badge for your app's icon on the iOS home screen:



If you pass in *0* as number, it will remove this badge. This is particularly useful if you want to clear a badge set by a push notification.

`notification.`**`setBadgeNumber`**(*number*, *success*, *error*)

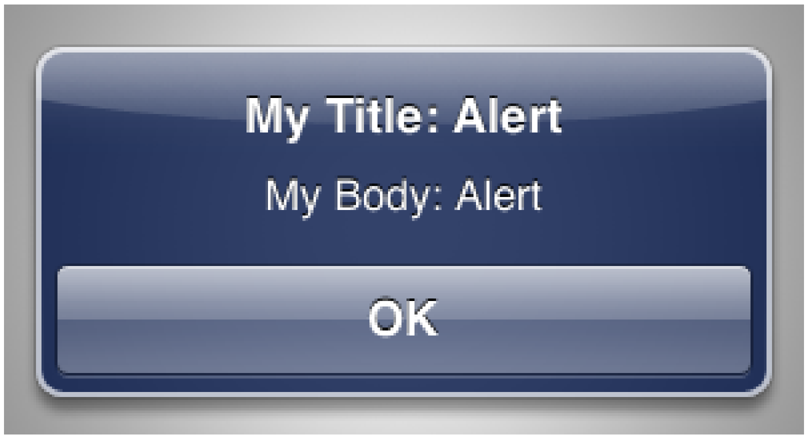
Arguments

- **number** (*string*) – number

`alert` **Platforms: Android, iOS**

Show a dialog window with text content that the user can dismiss. Similar to using `window.alert` except you can control the title text for the dialog.

To create an alert like this:



You would use this code:

```
forge.notification.alert("My Title: Alert", "My Body: Alert", function () {
  // ... implement logic for when user dismissed alert
});
```

`notification.`**`alert`** (*title*, *body*, *success*, *error*)

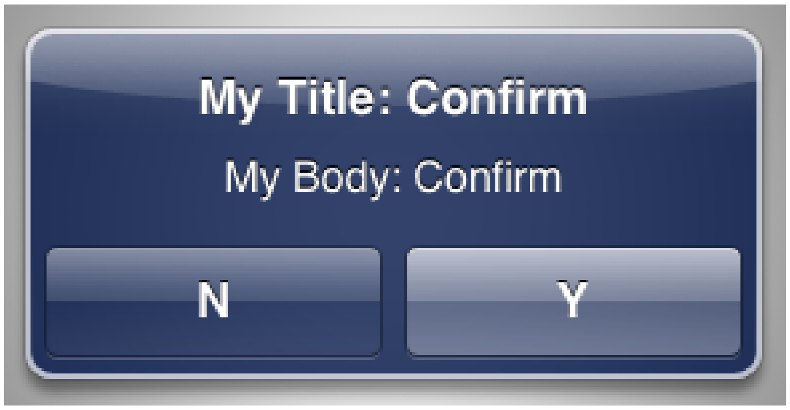
Arguments

- **title** (*string*) – text to show as the title of the dialog
- **body** (*string*) – text to show as the body of the dialog
- **success** (*function()*) – called when the user dismisses the alert
- **error** (*function(content)*) – called with details of any error which may occur

`confirm` **Platforms: Android, iOS**

Show a dialog window prompting the user with a “yes”/”no” style question.

To show a confirmation dialog like this:



You would use this code:

```
forge.notification.confirm("My Title: Confirm", "My Body: Confirm", "Y", "N", function (userClickedYe
  if (userClickedYes) {
    // ... implement logic for when user clicked "Y"
  } else {
    // ... implement logic for when user clicked "N"
  }
});
```

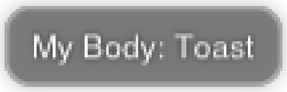
`notification.`**`confirm`**(*title*, *body*, *positive*, *negative*, *success*, *error*)

Arguments

- **title** (*string*) – text to show as the title of the dialog
- **body** (*string*) – text to show as the body of the dialog
- **positive** (*string*) – text to use for the positive action button
- **negative** (*string*) – text to use for the negative action button
- **success** (*function(result)*) – called with `true` if the user selected the positive option, `false` if they selected the negative option.
- **error** (*function(content)*) – called with details of any error which may occur

`toast` **Platforms: Android, iOS**

Create a small popup which disappears after a few seconds.



`notification.`**`toast`**(*body*, *success*, *error*)

Arguments

- **body** (*string*) – body
- **success** (*function()*) – called if the toast is displayed successfully
- **error** (*function(content)*) – called with details of any error which may occur

Permissions On Chrome this module will add the `notifications` permission to your app, users will be prompted to accept this when they install your app.

On Android this module will add the `VIBRATE` permission.

`package_names`: Built package names This controls the package name used internally when building your app; if you don't set this manually, we'll generate a package name automatically for you, which looks something like `io.trigger.forge7faf8ebcb8a111e1910212313d1adcbe`.

Although your users aren't really going to see this value, it can be useful to be able to control it manually, for example when updating a previous app that wasn't built on Forge, or if you already have iOS provisioning profiles which don't match our generated ID.

Config `package_names` should be an object mapping a target name onto a package name, e.g.:

```
"android": "com.example.my_app_name",
"ios": "com.example.ios.app"
```

Currently, only `android` and `ios` are supported.

```
{
    "modules": {
        "package_names": {
            "android": "com.example.app",
            "ios": "com.example.ios.app"
        }
    }
}
```

`parameters`: Custom config options `parameters` are custom config options your app can access via `forge.config.modules.parameters`.

Config

```
{
    "modules": {
        "parameters": {
            "my_option": "my data"
        }
    }
}
```

`payments`: In-app payments

Config The `payments` module must be enabled in `config.json`

```
{
    "modules": {
        "payments": {
            "androidPublicKey": "BASE64KEY=="
        }
    }
}
```

API The `forge.payments` namespace allows access to native in-app payment processing.

`transactionReceived.addListener` Platforms: Mobile

Triggered for all payment transaction events, if the payments module is used this listener must be add on all pages to handle transaction events.

The callback for this event will be invoked with two arguments, the first is an object containing the following keys:

- `orderId`: A unique order ID returned from iTunes or Google Play, use this to match refunds or cancellations to purchases.
- `productId`: The product involved in this transaction
- `purchaseTime`: The date and time of purchase

- `purchaseState`: One of `PURCHASED`, `REFUNDED`, `CANCELED` or `EXPIRED`. Gives the state of the transaction, only `PURCHASED` exists on iOS.
- `notificationId`: An internal identifier used by Forge when confirming transactions.
- `receipt`: Receipt information which can be sent to a server and verified with iTunes or Google Play. See the guide for more details.

The 2nd argument is a function used to confirm the transaction has been processed, this is required by both iOS and Android to confirm the transaction has been both recieved and correctly processed.

Example:

```
forge.payments.transactionReceived.addListener(function (data, confirm) {
  if (data.purchaseState == "PURCHASED") {
      alert("Thanks for buying: "+data.productId);
      confirm();
  } else {
      alert("Your product '"+productId+"' has been removed");
      confirm();
  }
});
```

`purchaseProduct` Platforms: Mobile

Purchase an in-app product identified by `productId`. The `success` callback will be called when the purchase has been approved by the user; after this the transaction information will be returned via the `transactionReceived` listener.

`payments.`**`purchaseProduct`**(*productId*, *success*, *error*)

Arguments

- **productId** (*string*) – product id registered with iTunes or Google Play.
- **success** (*function()*) – Purchase approved
- **error** (*function(content)*) – called with details of any error which may occur

`startSubscription` Platforms: Mobile

The same as `purchaseProduct` but starts a subscription rather than a one-off purchase.

`payments.`**`startSubscription`**(*productId*, *success*, *error*)

Arguments

- **productId** (*string*) – product id registered with iTunes or Google Play.
- **success** (*function()*) – Purchase approved
- **error** (*function(content)*) – called with details of any error which may occur

`restoreTransactions` Platforms: Mobile

If in-app products are managed by iTunes or Google Play, previous transactions may be requested at any time. This allows transactions to be restored on a new device or if the app has been reinstalled. The `success` callback will be called once the request is made; after this the transaction information will be returned via the `transactionReceived` listener.

`payments.`**`restoreTransactions`**(*success*, *error*)

Arguments

- **success** (*function()*) – Request sent
- **error** (*function(content)*) – called with details of any error which may occur

Guide

Using in-app payments

Javascript

- In order to receive asynchronous transaction details you must register a `transactionReceived` listener on all pages in your app: this must deal with incoming transactions and call the confirmation function when they are dealt with. Not calling the confirm function will cause transactions events to be emitted again at a later time as iTunes/Google Play will assume your app has not handled them successfully.
- To purchase a product use `forge.payments.purchaseProduct`. When calling `purchaseProduct` pass the ID of your product as created on iTunes/Google Play.
- To begin a subscription use `forge.payments.startSubscription`. Similarly to `purchaseProduct` pass in the product ID of the package you wish to subscribe to. See the `receipts` section below for details on verifying subscription status.

Android

When developing your app and not signing with a release key, you can use the following special product IDs to test in-app paym

- `android.test.purchased`: This product will return a successful `PURCHASED` transaction if the user presses “Buy”. It is not managed and so will not be restorable.
- `android.test.canceled`: This product will return a `CANCELED` transaction immediately if the user presses “Buy”.
- `android.test.refunded`: This product acts the same as `android.test.canceled` but is marked as `REFUNDED`.
- `android.test.item_unavailable`: This product cannot be bought and will display an error to the user.

Important: In a real purchase, a `CANCELED` transaction may be returned after a `PURCHASED` transaction: in this situation, your app should be able to deal with revoking any features enabled by a previously `PURCHASED` transaction. If a transaction is cancelled it will have the same `orderId` as the original purchase.

Test products are a close simulation to actual products bought through in-app payments, but they do act in subtly different ways: it is important you also test your app with real purchases before deploying it to users.

In order to test your actual products you will need to make sure you have done the following things:

- Add your in-app products on the Google Play Developer Console. To do this you will need to sign up for a merchant account through the console.
- Make sure your in-app products are marked as published: unpublished products will not appear for test users in unpublished APKs

- Add test accounts in your profile on the Google Play Developer Console. As a developer you cannot purchase your own products, they must be purchased by a test account, who must be the primary user on the device you are testing on. If you (the developer) are the primary user on your device you will need to perform a factory reset and sign in with a test account to test your app.
- Copy the public key from your profile on the Google Play Developer Console into your payments module config.
- Package your app through Forge and upload the APK you wish to test to Google Play. You do not need to publish the app to test it as one of the test users you previously created, but you do need to upload it to enable in-app purchases.
- Install the APK you uploaded to Google Play to the device you wish to test on (and make sure your primary account on the device is a test user who has been added on Google Play).
- You should now be able to perform in-app purchase actions in your app. Test user purchases will be charged if you allow them to go through: you can manually cancel or refund purchases through the merchant account section of Google Play.

Important: You cannot buy your own products: test users must have a different ID to your merchant ID, and the test user must be the **primary** user account on the testing device.

Note: You cannot use the emulator to test in-app payments: it must be a real device.

Note: When uploading APKs and adding test users, we've found there can sometimes be a delay for the changes to take effect. If you see unexpectedly see messages like **this app is not configured for billing** or **this item is not available**, try waiting for 10 minutes.

iOS When developing on iOS, there are no test product IDs - only actual products created for your app in iTunes Connect can be tested. However, apps signed with a "iPhone Developer" certificate will run in the iTunes sandbox and any purchases will be simulated (no charge will be made).

In order to test in-app payments on iOS you must make sure you have completed the following steps:

- Create a specific app ID for your app in the iOS provisioning portal, and create development and distribution provisioning profiles for that app. Wildcard provisioning profiles will not work with in-app purchases.
- Add your app to iTunes Connect and add any in-app products you want to sell.
- *Package your app* with the *distribution* provisioning profile into an IPA and submit it to iTunes Connect; if you do not wish you submit your app for approval yet you can submit it then immediately reject the binary through iTunes Connect.
- Run the app on a device using the *development* provisioning profile to be able to test in the sandbox with dummy transactions.
- You cannot buy apps using a real iTunes account while testing: in order to test, you must sign out of the App Store on your device, and when using your app and prompted to login, sign in with a test user created through iTunes Connect.
- You may need to wait several hours between submitting your app and in-app items and them being available for you to test with. If you have followed all of the above steps and still have problems you may just need to wait for the changes you have made to become active.

Note: You can configure the provisioning profile and developer certificate to use in your `local_config.json` file, see *Configuration for the tools*. Being able to switch between development and distribution environments with *Profiles* is a time saver.

Managed products / `restoreTransactions` If you create "managed" items on Google Play or "Non-Consumable" items on iTunes Connect (this includes subscriptions on both platforms) then you can restore purchases the user has made at a later date, if they have reinstalled your app or moved to another device.

To restore transactions made on another install or another device use `forge.payments.restoreTransactions`, calling this may cause the user to be prompted for login details, so it is best to only call it when first setting up an application, or if a user specifically requests it. Any restored transactions will be returned through the `transactionReceived` listener.

Receipts In order to confirm a purchase has been legitimately made through iTunes or Google Play it is best to forward details of the transaction to your server and verify the transaction there. To allow this both iTunes and Google Play provide signed receipts for the transactions.

Android On Android, the `receipt` property of the transaction contains the `type` as `android`, as well as a `data` property containing a JSON string with the receipt data, a `purchaseToken` which can be used to verify subscription purchases, a `signature` property containing a base64 encoded signature and a `signed` property which is a boolean indicating whether or not the signature matches. Details on how to verify the signature can be found in the Android documentation: http://developer.android.com/guide/market/billing/billing_integrate.html#billing-signatures.

The `signed` property is determined on the device in Java and should not be trusted if the data can be sent to a server to be verified.

To verify subscription purchases,find out when they will expire and cancel subscriptions use the android-publisher API: https://developers.google.com/android-publisher/v1/

iOS On iOS the `receipt` property of the transaction contains the `type` as `iOS` and a `data` property which is a base64 encoded receipt. You can forward the receipt to iTunes in order to verify it by following the instructions provided by Apple: http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/VerifyingStoreReceipts/Verifying

Details on subscriptions and how to verify subscription receipts can also be found in the iOS documentation: http://developer.apple.com/library/ios/#documentation/NetworkingInternet/Conceptual/StoreKitGuide/RenewableSubscriptions/Renewa

`prefs`: Preferences Preferences are used to save state in your extension. State is persisted between restarts and is consistent across all parts of your app.

Config The `prefs` module must be enabled in `config.json`

```
{
    "modules": {
        "prefs": true
    }
}
```

API

`get`

`prefs.get` (*name*, *success*, *error*)

Arguments

- **name** (*string*) – the name of the preference you want to query
- **success** (*function(value)*) – will be invoked as the preference value as its only parameter
- **error** (*function(content)*) – called with details of any error which may occur

Platforms: All

If the preference has not been set (with *set*), and there is no default value for this preference, `null` is returned.

`set` Platforms: All

`prefs.set` (*name*, *value*, *success*, *error*)

Arguments

- **name** (*string*) – the name of the preference you want to save
- **value** – the value to save
- **success** (*function()*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

The preference value given here will override a default value (if one was given).

`clear` Platforms: All

`prefs.clear` (*name*, *success*, *error*)

Arguments

- **name** (*string*) – the name of the preference to un-set
- **success** (*function()*) – a callback to be invoked (with no arguments) when the operation is complete
- **error** (*function(content)*) – called with details of any error which may occur

Un-sets the given preference name, so that future calls to *set* will return `undefined` (or the default preference value, if given).

`clearAll` Platforms: All

`prefs.clearAll` (*success*, *error*)

Arguments

- **success** (*function()*) – a callback to be invoked (with no arguments) when the operation is complete
- **error** (*function(content)*) – called with details of any error which may occur

Un-sets all preference names, so that calls to *set* will return `undefined` (or the default value for a preference, if given).

keys **Platforms: All**

`prefs.keys(success, error)`

Arguments

- **success** (*function(keysArray)*) – invoked with an array of the set key names as its only argument
- **error** (*function(content)*) – called with details of any error which may occur

Find which preferences have been set.

push: notifications (with Parse) "Add a Backend to Your Mobile App in Minutes" - Parse allows you to add backend features to your mobile app without a server. Their back end features include a datastore, user accounts and push notifications.

For a small example on our blog, see Using Parse and Trigger.io for cross-platform apps without pain in the back-end.

Parse push notifications are integrated directly Forge. Other Parse features may be accessed by using *forge.request.ajax* with the Parse REST API.

Configuring Push Notifications First you'll need to register an app at parse.com.

In order for your app to communicate with the Parse servers for push notifications you must specifiy both your applicationId and clientKey in your app's *config.json* as shown:

```
"partners": {
    "parse": {
        "applicationId": "YourApplicationKeyZdAtirdSn02Qy6NofiU2Hf",
        "clientKey": "YourClientKeyZdAtirdSn02QQy6NofiU2Hfy6No"
    }
}
```

Note: This Parse configuration must go in the *partners* section of your `config.json`, not *modules* as is the norm with normal Forge modules.

API: forge.partners.parse Push notifications received through Parse can be used with the generic push notification event in Forge, see the *event API* for more details. The following code is an example of how to show an alert to a user when a push notification is received.

Example:

```
forge.event.messagePushed.addListener(function (msg) {
    alert(msg.alert);
});
```

You can try out sending a push notification from your app's control panel at parse.com.

Parse uses channels to send push notifications to specific groups of users. By default all users are subscribed to the empty channel; if you wish to send push notifications to specific users, you can use the following methods to manage which channels a user is subscribed to.

installationInfo **Platforms: Mobile**

Every Parse installation has a unique ID associated with it; you can use this method to retrieve the installation ID for this user and do things like advanced targetting of push notifications.

`parse.`**`installationInfo`**(*success*, *error*)

Arguments

- **success** (*function(info)*) – Called if the request is successful: `info` will contain at least an `id` entry
- **error** (*function(content)*) – Called with details of any error which may occur

Example:

```
forge.partners.parse.installationInfo(function (info) {
    forge.logging.info("installation: "+JSON.stringify(info));
});
```

push.subscribe **Platforms: Mobile**

`parse.push.`**`subscribe`**(*channel*, *success*, *error*)

Arguments

- **channel** (*string*) – Identifier of the channel to subscribe to
- **success** (*function()*) – Called if the request is successful
- **error** (*function(content)*) – Called with details of any error which may occur

Example:

```
forge.partners.parse.push.subscribe("beta-testers",
function () {
  forge.logging.info("subscribed to beta-tester push notifications!");
},
function (err) {
  forge.logging.error("error subscribing to beta-tester notifications: "+
    JSON.stringify(err));
});
```

push.unsubscribe **Platforms: Mobile**

`parse.push.`**`unsubscribe`**(*channel*, *success*, *error*)

Arguments

- **channel** (*string*) – Identifier of the channel to unsubscribe from
- **success** (*function()*) – Called if the request is successful
- **error** (*function(content)*) – Called with details of any error which may occur

Example:

```
forge.partners.parse.push.unsubscribe("beta-testers",
function () {
  forge.logging.info("no more beta-tester notifications...");
},
function (err) {
  forge.logging.error("couldn't unsubscribe from beta-tester notifications: "+
    JSON.stringify(err));
});
```

push.subscribedChannels Platforms: Mobile

`parse.push.subscribedChannels(success, error)`

Arguments

- **success** (*function(channels)*) – Called with an array of subscribed channels
- **error** (*function(content)*) – Called with details of any error which may occur

Example:

```
forge.partners.parse.push.subscribedChannels(
function (channels) {
  forge.logging.info("subscribed to: "+JSON.stringify(channels));
},
function (err) {
  forge.logging.error("couldn't retreive subscribed channels: "+
    JSON.stringify(err));
});
```

Permissions On Android this module will add the `VIBRATE` and `RECEIVE_BOOT_COMPLETED` permissions to your app, users will be prompted to accept this when they install your app.

`reload`: Push updates to deployed apps The Reload module allows you to update the HTML, CSS and JavaScript in your app for your users without you needing to push an App Store update or your users needing to re-install the app.

Config The `reload` module must be enabled in `config.json`, once the `reload` module is enabled you will be able to push updates to any builds of your app from that point.

```
{
    "modules": {
        "reload": true
    }
}
```

Concepts You can reload updates via the Toolkit UI, *command-line* or *standalone build API*.

You can reload all of your users, or divide them into streams for A/B testing.

You can use the default update behavior without altering your code at all beyond including the reload module in the configuration. Or you can use the JavaScript API described here to have finer grain control over how reloads occur in your app.

You can Reload your app with up to 250Mb of content via Trigger.io. You can Reload your app with larger content hosted in a *3rd party CDN like Rackspace Cloud Files*.

How Reloads work When you reload a new version of your code through the Toolkit UI, users will see the new version of your code when they switch away from your app and switch back to it: we download updated files when focus is lost and apply the changes when focus is restored.

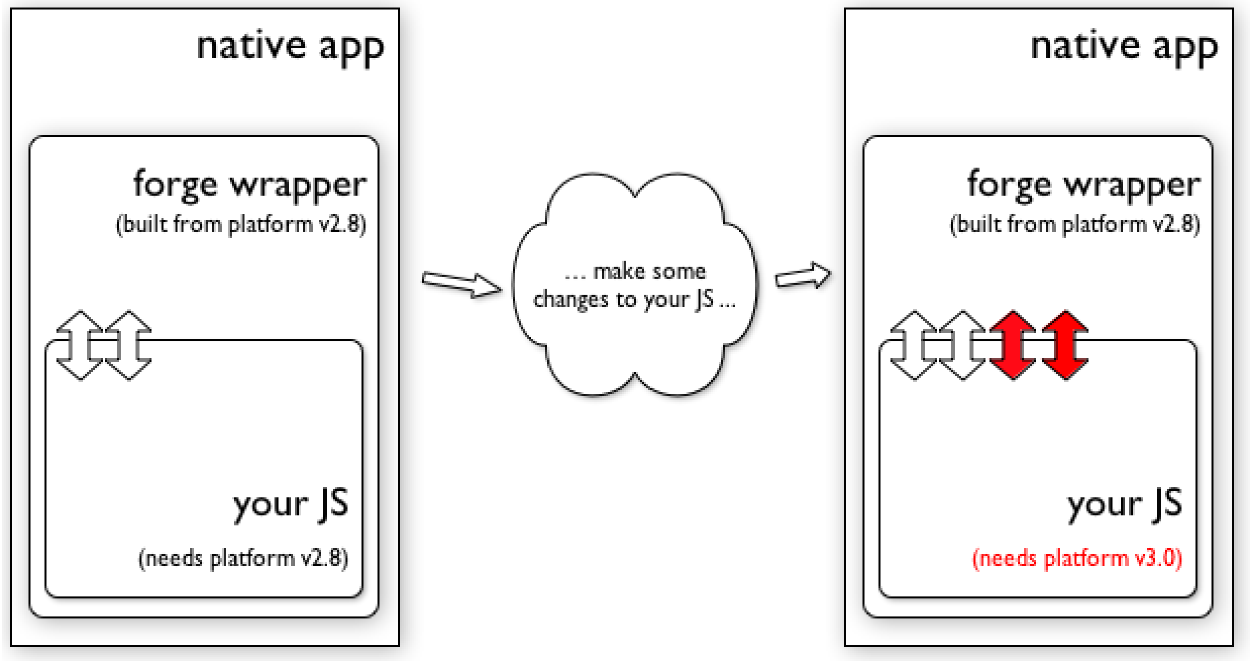
You have control over this process using the JavaScript API described here, but the default behavior does not require any change to your app's logic.

Streams Streams let you segment your user base so that not everyone has to receive every update you push with Reload. This is useful for pushing regular unstable updates to developers, release candidates to beta testers and stable versions to the wider user base.

By default, your users are added to the "default" stream - you can create as many streams as you want, and switch apps to different streams using using the `switchStream` method described below.

Config versions

You must only reload HTML / CSS and JavaScript where it works with the minor version of the Forge platform which you originally built against.



If your updated JavaScript relies on Forge APIs that are only available in newer versions of the wrapper than your users currently have installed, then you will need to re-package your apps and deploy through the app stores again.

API In order to use `reload` no API calls are required, however some apps may which to use the following API methods to force updates at a users request, or switch the user to an alternate streams.

`updateAvailable` Platforms: Mobile

Gives `true` or `false` depending on whether there is a `reload` update available to be downloaded.

`reload.`**`updateAvailable`**`(`*success*, *error*`)`

Arguments

- **success** (*function(update_available)*) – Called with whether or not an update is available.
- **error** (*function(content)*) – called with details of any error which may occur

`update` Platforms: Mobile

Forces the application to check for, and if available download a `reload` update. The update will then be applied the next time the app is closed or loses focus.

If you have previously called `pauseUpdate`, this will override that and continue Reload updates - including any in-progress updates if applicable.

`reload.`**`update`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – Called when an update is available and the download has started - see the `updateReady` event to be notified when the update is complete
- **error** (*function(content)*) – called with details of any error which may occur

`pauseUpdate` Platforms: Mobile

If there is a Reload update in-progress (i.e. downloading files), we will halt downloading after the current file has been received.

Reload updates will not be downloaded for this app until you explicitly call `update`. Re-installing the app via the Toolkit (during development) or via an app store (in production) will also re-enable Reload updates.

`reload.`**`pauseUpdate`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – called when the Reload updates have been successfully paused
- **error** (*function(content)*) – called with details of any error which may occur

`applyNow` Platforms: Mobile

Warning: This method is deprecated and does not do anything any more. Reload updates can only be applied by the user closing or switching from the app. This is due to a bug in iOS's webkit implementation.

`switchStream` Platforms: Mobile

Switches the `reload` stream the app will download updates from.

`reload.`**`switchStream`**`(`*stream_name*, *success*, *error*`)`

Arguments

- **success** (*function()*) – Stream switched
- **error** (*function(content)*) – called with details of any error which may occur

`updateReady.addListener` Platforms: Mobile

Fired when a Reload update has been downloaded and is ready to apply.

`reload.updateReady.`**`addListener`**`(`*callback*, *error*`)`

Arguments

- **callback** (*function()*) – an update will be applied next time the app resumes
- **error** (*function(content)*) – called with details of any error which may occur

Update process The `reload` update process has several parts. First, it must be determined if an update is available, and if it is available it needs to be downloaded. Once an update has been downloaded it has to be applied, this means making the new files available to the app. If the app is running while an update is applied then there may need to be additional code in the app to make use of the updated files.

The following things will cause `reload` to download new update files if available:

- A call to `forge.reload.update()`.
- On all platforms new files will be downloaded shortly after the app is launched.
- On Android and iOS new files will also be downloaded when the app loses focus but is running in the background.
- On Android new files will also be downloaded when the app exits.

Assuming an update has been fully downloaded and is ready to apply the following things will replace the apps assets files with the new update:

- On all platforms when the app is relaunched (i.e. when it has been quit and opened again).
- On iOS and Android when the app is restored from the background.

If updates are applied during launching or restoring an app `index.html` will be reloaded with the new update files.

Notes

- See *Using Trigger.io Reload* for more information about how to use Reload
- Updates may take some time if the user is on a slow network, however several things are done to improve this, only changed files are downloaded in an update, and if an update is interrupted part way through it will resume where it left off next time it is started.
- On iOS updates are given 10 minutes to download each time the app is paused as this is the maximum amount of background processing time available on iOS. If an update is interrupted it will resume where it left off on the next attempt.
- Only one update is downloaded at a time, if an update is waiting to be applied any future updates will not be downloaded until it has been applied to the app. This should never be a problem for real users but may be confusing during testing.
- When testing the easiest way to cause an update is to leave the app by pressing the home button on the device, wait a few seconds (or look at the log output to see when the reload update is complete), and reopen the app to see the update applied.

Important: Currently, pushing a Reload update will cause iOS devices to copy files out of the installed app bundle into a secondary area. This is due to sandboxing rules which prevent us directly accessing files in your installed app after a Reload has been completed. In order to avoid your app taking too much storage space on the device, it's recommended you distribute larger files using something like *forge.file.saveURL*, rather than including them in the app bundle.

We are actively looking for a way to avoid this limitation.

`request`: Cross-domain requests Normal in-page JavaScript is only able to make HTTP requests to the same server as one hosting the current web page. These methods allow you to work around this restriction.

Config

Note: For security reasons, you must specify the remote URLs you will send requests to in the `permissions` array of your JSON configuration file.

This array, like the `patterns` array, should contain Match Patterns to match the remote URLs you want to interact with.

To protect your users, make these match patterns as restrictive as possible.

```
{
    "modules": {
        "request": {
            "permissions": ["https://trigger.io/*"]
        }
    }
}
```

API

`get` Platforms: All

`request.get` (*url*, *callback*, *error*)

Arguments

- **url** (*string*) – the URL to GET
- **callback** (*function(content)*) – called with the retrieved content body as the only argument
- **error** (*function(content)*) – called with details of any error which may occur

The callback function *callback* is invoked with the content body of the requested URL. JSON-encoded content will automatically be parsed into a JavaScript object.

As it is limited to `GET` requests and lacks the more advanced options of *forge.request.ajax*, it's recommended that *forge.request.get* is only used in very simple scenarios.

`ajax` Platforms: All

`request.ajax` (*options*)

Arguments

- **options** (*object*) – jQuery-style parameters to control the request

Note: unlike jQuery, we expect the URL for the the request to be passed into the options hash, *not* as a positional parameter

This function is closer to the jQuery.ajax method than *forge.request.get*. However, the full range of jQuery options are **not supported** for this method, due to the structure of browser and mobile apps.

Also, note that the `error` and `success` callbacks are **not** passed a jQuery XHR object.

Currently supported options:

- accepts

- cache
- contentType
- data
- dataType
- error
- password
- success
- timeout
- type
- url
- username
- files (Mobile only, see *forge.file*)
- fileUploadMethod
- headers

Error object properties:

- `statusCode`: Status code returned from the server.
- `content`: Content returned from the server (if available).

Error values (see *error callback docs* for more detail):

- `type`: `"UNAVAILABLE"`
- `subtype`: `"NO_INTERNET_CONNECTION"` No internet connection is currently available (on iOS it is required you inform the user of this if it impacts their current experience).

Example:

```
forge.request.ajax({
  type: 'POST',
  url: 'http://my.server.com/update/',
  data: {x: 1, y: "2"},
  dataType: 'json',
  headers: {
    'X-Header-Name': 'header value',
  },
  success: function(data) {
    alert('Updated x to '+data.x);
  },
  error: function(error) {
    alert('Failed to update x: '+error.message);
  }
});
```

You can control the name of uploaded files by setting the `name` attribute, e.g.:

```
myFile.name = 'name_of_input';
forge.request.ajax({
  type: 'POST',
  url: 'http://my.server.com/upload/',
  files: [myFile],
  success: function(data) {
    alert('Uploaded file as '+myFile.name);
  },
  error: function(error) {
    alert('Failed to upload file: '+error.message);
  }
});
```

If you need to POST an image as the whole request body, use `fileUploadMethod`. E.g.:

```
forge.request.ajax({
  type: 'POST',
  url: 'http://my.server.com/upload_image/',
  fileUploadMethod: "raw",
  success: function(data) {
    alert('Uploaded image');
  }
});
```

In this example, the `Content-Type` header will be set to `image/jpeg` and the POST body will consist of just the image data with no extra encoding. This is useful in conjunction with services like Parse.

Permissions On Chrome this module will any of the Match Patterns you specify to your app, users will be prompted to accept this when they install your app.

`requirements`: App requirements The `requirements` module allows you to set specific device requirements for your apps.

Config

```
{
    "requirements": {
        "android": {
            "minimum_version": "6",
            "disable_ics_acceleration": true
        },
        "ios": {
            "minimum_version": "4.3",
            "device_family": "iphone"
        },
        "chrome": {
            "content_security_policy": "script-src 'self' https://ssl.google-analytics.com; object-s
            "web_accessible_resources": [
                "background.jpg"
            ]
        }
    }
}
```

Android

- `minimum_version`: the minimum Android API level you want to support, it must be between 5 and 15. More details can be found on the Android developers site: http://developer.android.com/guide/appendix/api-levels.html.

- `disable_ics_acceleration`: Disables hardware acceleration on Android 4.0, this is a workaround to potential rendering issues which can affect some apps on this particular version of Android.

iOS

- `minimum_version`: The iOS version is the minimum iOS version you want to support, between 4.0 and 5.1.
- `device_family`: Used to limit the types of device which can run the app, must be one of `any`, `iphone` or `ipad`.

Chrome The Chrome options relate to Chromes "manifest_version": 2 changes, which are documented on the Chrome website http://code.google.com/chrome/extensions/manifestVersion.html. The available settings are:

- `content_security_policy`: This determines what javascript can be executed in pages that belong to your extension (such as popups). The example given above is how you would allow Google Analytics to work within an extension. More documentation is available on the Chrome site http://code.google.com/chrome/extensions/contentSecurityPolicy.html
- `web_accessible_resources`: This is an array of any files in your extension which are to be accessed from external sites. The most common use of this is if your extension uses a content script to load images from your extension into a 3rd party site.

`sms`: SMS messaging The `forge.sms` namespace allows you to prompt the user to send SMS messages from your app.

Config The `sms` module must be enabled in `config.json`

```
{
    "modules": {
        "sms": true
    }
}
```

API

`send` Platforms: Mobile

Send an SMS message, optionally with one or more recipients and a message pre-filled in.

Example:

```
forge.sms.send({
  body: "Hello, World!",
  to: ["123457869", "328592835"]
}, function () {
  alert("Message sent");
});
```

`sms.`**`send`**(*params*, *success*, *error*)

Arguments

- **params** (*function()*) – Data to pre-fill SMS message with, contains `body` which is the content of the message, and `to` as either a string or array of phone numbers to send to.
- **success** (*function()*) – callback to be invoked when no errors occur

- **error** (*function(content)*) – called with details of any error which may occur

`tabbar`: Native tab bar The `tabbar` module shows a fixed footer as part of your app which can show multiple "tab" buttons, these buttons can be selected by the user or activated by Javascript.

To get an idea of how these footers can look, see our blog post, How to build hybrid mobile apps combining native UI components with HTML5.

Config The `tabbar` module must be enabled in config.json as follows:

```
{
    "modules": {
        "tabbar": true
    }
}
```

API

`show` Platforms: Mobile

Shows the tabbar. The tabbar is shown by default and will only be hidden if you call `tabbar.hide()`.

`tabbar.`**`show`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`hide` Platforms: Mobile

Hides the tabbar.

`tabbar.`**`hide`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setTint` Platforms: Mobile

Set a colour to tint the tabbar with, in effect the tabbar will become this colour with a gradient effect applied.

`tabbar.`**`setTint`**`(`*color*, *success*, *error*`)`

Arguments

- **color** (*array*) – an array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255].
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setActiveTint` Platforms: Mobile

Set a colour to tint active tabbar item with.

`tabbar.`**`setActiveTint`**`(`*color*, *success*, *error*`)`

Arguments

- **color** (*array*) – an array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255].
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`addButton` Platforms: Mobile

Add a button with an `icon` and `text` to the tabbar. The first parameter is an object of options for the button, which can include:

- `icon` (required): This sets the icon which will be shown on the tab, only the alpha channel of the icon will be used with the color being replaced for a consistent style. You can use a file object as returned by something like *forge.file.saveURL*, or a string path relative to the `src` directory, e.g. `"img/button.png"`.
- `text` (required): This sets the text which will appear below the icon on the tab.
- `index` (recommended): This sets the order of the button to be added, not setting this will result in the order of the tabs not being fixed.

When adding a button the success callback will be called with a button object. This object has methods to allow you to interact with the button as follows:

- `button.remove(success, error)`: Remove the button
- `button.setActive(success, error)`: Mark the button as selected, without calling this and before the user clicks on one of the buttons no button will be marked active.
- `button.onPressed.addListener(callback, error)`: Add a callback to handle when the button is pressed

Example:

```
forge.tabbar.addButton({
  icon: "search.png",
  text: "Search",
  index: 0
}, function (button) {
  button.setActive();
  button.onPressed.addListener(function () {
    alert("Search");
  });
});
```

`tabbar.`**`addButton`**`(`*params*, *success*, *error*`)`

Arguments

- **params** (*object*) – Button options, must contain an `icon`, `text` and optionally `index`
- **success** (*function(button)*) – called with the button object.
- **error** (*function(content)*) – called with details of any error which may occur

`removeButtons` Platforms: Mobile

Remove all buttons from the tabbar.

`tabbar.removeButtons` (*success*, *error*)

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setInactive` Platforms: Mobile

Unselect any currently active tab, leaving the tabbar with no tabs selected.

`tabbar.setInactive` (*success*, *error*)

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`tabs`: Tabs Management

Tabs management provides functionality specific to tabs.

Config

The `tabs` module must be enabled in `config.json`

```
{
    "modules": {
        "tabs": true
    }
}
```

API

`open` Platforms: All except web

Opens a new tab with the specified url with an option to retain focus on the calling tab.

On mobile this will display a *modal view* and the success callback will be called with an object containing a url and optionally a userClosed boolean property.

`tabs.open` (*url*[, *keepFocus*], *success*, *error*)

Arguments

- **url** (*string*) – The URL to open in the new tab
- **keepFocus** (*boolean*) – (optional) If true keeps the current tab focused
- **success** (*function(object)*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

`openWithOptions` Platforms: All except web

As `open` but takes an object with the following parameters:

Required:

- `url`: Required URL to open

Browser only:

- `keepFocus`: Whether or not to keep focus on the current page.

Mobile only:

- `pattern`: Pattern to close the modal view, see *modal views* for more detail.
- `title`: Title of the modal view.
- `tint`: Colour to tint the top bar of the modal view. An array of four integers in the range [0,255] that make up the RGBA color. For example, opaque red is [255, 0, 0, 255].
- `buttonText`: Text to show in the button to close the modal view.
- `buttonIcon`: Icon to show in the button to close the modal view, if `buttonIcon` is specified `buttonText` will be ignored.
- `buttonTint`: Colour to tint the button of the top bar in the modal view.

Example:

```
forge.tabs.openWithOptions({
  url: 'http://my.server.com/login/',
  pattern: 'http://my.server.com/loggedin/*',
  title: 'Login Page'
}, function (data) {
  forge.logging.log(data.url);
});
```

`tabs.`**`openWithOptions`**`(`*options*, *success*, *error*`)`

Arguments

- **options** (*object*) – Object containing url and optional properties.
- **success** (*function(object)*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

`closeCurrent` Platforms: Browser only

Restrictions: Only available inside a page (not from a background script)

Close the tab which makes the call.

`tabs.`**`closeCurrent`**`(`*error*`)`

Arguments

- **error** (*function(content)*) – called with details of any error which may occur

Permissions On Chrome this module will add the `tabs` permission to your app, users will be prompted to accept this when they install your app.

`tools`: Miscellaneous tools

Config The `tools` module must be enabled in `config.json`

```
{
    "modules": {
        "tools": true
    }
}
```

API

`UUID` Platforms: All

A UUID is a globally unique token; when represented as a string, they look something like `18ADF182-7B12-4FA1-AF0B-6032108C0AE8`. Forge already uses UUIDs internally to ensure your extension doesn't conflict with others; this method returns a new UUID for you to use as a unique token.

Note: This function is synchronous and returns a value rather than taking a callback.

`tools.`**`UUID`**`()`

Returns a string representation of the UUID

`getURL` Platforms: All

Resolve this name to a fully-qualified local or remote resource

`tools.`**`getURL`**`(`*name*, *callback*, *error*`)`

Arguments

- **name** (*string*) – unqualified resource name, e.g. `my/resource.html`
- **callback** (*function(url)*) – will be invoked with the URL as its only parameter
- **error** (*function(content)*) – called with details of any error which may occur

`topbar`: Native top bar The `topbar` module displays a fixed native header bar in mobile apps and provides a Javascript API to modify it at runtime.

To get an idea of how these headers can look, see our blog post, How to build hybrid mobile apps combining native UI components with HTML5.

Config The `topbar` module must be enabled in config.json as follows:

```
{
    "modules": {
        "topbar": true
    }
}
```

Style Guidlines The `setTitleImage` method and `icon` option in the `addButton` method allow you to specify images within the topbar element. These guidelines may help you to make them look good across devices:

- The title image will be scaled down (but not up) to fit the height of the topbar exactly. This means any padding should be included in the image, and the image should be at least 100px high.

- The button icons will also be scaled down (but not up) to fit the height of the button precisely. The width of the button is the width of the icon (or text) plus a small amount of padding. We'd recommend button icons are at least 64px high to make sure they always fill the button.
- The total width of the title and buttons is not checked by Forge, so its up to you to test everything fits, We'd recommend leaving spare space to make sure devices with unexpected screen ratios don't overlap.

API

`show` Platforms: Mobile

Shows the topbar. The topbar is shown by default and will only be hidden if you call `topbar.hide()`.

`topbar.`**`show`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`hide` Platforms: Mobile

Hides the topbar.

`topbar.`**`hide`**`(`*success*, *error*`)`

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setTitle` Platforms: Mobile

Set the title displayed in the top bar.

`topbar.`**`setTitle`**`(`*title*, *success*, *error*`)`

Arguments

- **title** (*string*) – Title to be displayed
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setTitleImage` Platforms: Mobile

Set the title displayed in the top bar to an image.

`topbar.`**`setTitleImage`**`(`*image*, *success*, *error*`)`

Arguments

- **image** (*file*) – file object as returned by something like *forge.file.saveURL*, or a string path relative to the `src` directory, e.g. `"img/button.png"`
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setTint` **Platforms: Mobile**

Set a colour to tint the topbar with, in effect the topbar will become this colour with a gradient effect applied.

On iOS 6 this color will also be used to tint the status bar, you can use this in combination with hiding the topbar if you only want a colored status bar and not a topbar.

`topbar.`**`setTint`**(*color*, *success*, *error*)

Arguments

- **color** (*array*) – an array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255].
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`addButton` **Platforms: Mobile**

Add a button with an icon to the top bar. The first parameter is an object describing the button with the following properties:

- `icon`: An icon to be shown on the button: this should be relative to the `src` directory, e.g. `"img/button.png"`.
- `text`: Text to be shown on the button, either `text` or `icon` must be set.
- `type`: Create a special type of button, the only option currently is `"back"` which means the button will cause the webview to go back when pressed.
- `style`: Use a predefined style for the button, currently this can either be `"done"` which will style a positive action (which may be overriden by `tint`), or `"back"` to show a back arrow style button on iOS.
- `position`: The position to display the button, either `left` or `right`. If not specified the first free space will be used.
- `tint`: The color of the button, defined as an array similar to `setTint`.

Example:

```
forge.topbar.addButton({
  text: "Search",
  position: "left"
}, function () {
  alert("Search pressed");
});
```

`topbar.`**`addButton`**(*params*, *callback*, *error*)

Arguments

- **params** (*object*) – Button options, must contain at least `icon` or `text`
- **callback** (*function()*) – callback to be invoked each time the button is pressed
- **error** (*function(content)*) – called with details of any error which may occur

`removeButtons` **Platforms: Mobile**

Remove currently added buttons from the top bar.

`topbar.`**`removeButtons`**(*success*, *error*)

Arguments

- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`urlhandler`: Custom URL schemes

Config The `urlhandler` module must be enabled in `config.json`

```
{
    "modules": {
        "urlhandler": {
            "scheme": "your.url.scheme"
        }
    }
}
```

API

`urlhandler.urlLoaded` Platforms: Mobile

Triggered when the app is loaded by a custom URL scheme.

`urlhandler.urlLoaded.`**`addListener`**(*callback*, *error*)

Arguments

- **function({url** – url}) callback: callback invoked with details of the url.
- **error** (*function(content)*) – called with details of any error which may occur

`ui`: Native UI enhancements This module includes miscellaneous native UI enhancements. This currently consists of enhancement to date, time and datetime form elements on Android.

Config The `ui` module must be enabled in `config.json`

```
{
    "modules": {
        "ui": true
    }
}
```

API

`enhanceInput` Platform: Android

This method can be used to enhance a specific or set of specific input elements, when called any input elements of type `date`, `time`, `datetime` or `datetime-local` matching the given selector will be updated to show a native date picker when the user taps on them. The value the user selects will be inserted into the input element so it can be treated as normal by other Javascript. The input element will also be disabled to prevent other user input into the field.

After the user selects a value from a picker the `blur` event will be triggered on the input element. This is the same behaviour as the native pickers built into iOS.

`ui.`**`enhanceInput`**(*selector*)

Arguments

- **selector** (*string*) – Any input elements of type date, time or datetime matching this selector will be enhanced.

`enhanceAllInputs` **Platform: Android**

This method will enhance all appropriate input elements currently in the DOM.

`ui.enhanceAllInputs()`

Native plugins

Important: Plugins are currently a beta feature, get in touch with support@trigger.io if you're interested in trying them out.

Native plugins allow you to write native Android and iOS code, compile it and include this compiled code in your app built using Forge. As part of this it also provides a method for your native code to communicate with your Javascript code.

Writing native plugins is obviously different for Android and iOS, however we have tried to keep the experience as consistent as possible. With that in mind these docs are split into the various tasks you might want to complete as part of writing a native plugin, with instructions for Android and iOS included in each section.

The Basics In order to allow developers to write native plugins we provide what we call **inspector projects** for both Android and iOS. These are Eclipse and Xcode projects that include the core library used by Forge, as well as an example plugin to get you started. This environment can be used as a sandbox to develop and compile your plugin.

Once you are happy with your plugin you can use the Toolkit to upload a compiled version with any additional files (detailed below) to the Forge build server. Once we have a copy of your plugin you can then add the plugin to your app through the app's config in the Toolkit.

Downloading the inspector projects Getting setup to develop native plugins is fairly simple, but there are some prerequisites you need to install.

Android

1. You will need Eclipse setup with the Android SDK, you can find details on how to install this on the Android developer site: http://developer.android.com/sdk/installing/index.html
2. Download the ForgeInspector project through the Toolkit. You should be able to do this from the plugin's toolkit page once you have a local template setup. If you can't see plugins in the Toolkit, then contact us at support@trigger.io to get into the native plugins beta.
3. `ForgeCore`, `ForgeInspector` and `ForgeTemplates` should have been downloaded to the `inspector/an-inspector` directory
4. In Eclipse, go to `File` then `Import Project...` and import the `an-inspector` directory: all three projects will appear in Eclipse

iOS

1. You will need a Mac running OS X
2. You will need Xcode 4.5, this is available as a free download in the OS X App Store
3. Download the ForgeInspector project through the plugin's page in the Trigger Toolkit.

4. Open this project in Xcode.

At this point you can run the inspector project. It includes an app which can list native methods exposed to Javascript and allows you to call them with sample data. It also includes a small example plugin called `alert` which allows you to show native alert style dialogs, and triggers an event in Javascript when the app is paused and resumed. It is probably a good idea to try out this plugin to get a feel for the layout of a plugin before you get started.

Updating inspector projects Some plugins will need to make changes to the Forge build process, such as including 3rd party frameworks, registering a URL handler or requesting app permissions. In order to do this, *additional steps* can be included in a plugin which will be run when the plugin in used in an app.

When you update your plugin's build steps or change the `platform_version` in manifest.json, your inspector project will need to be updated.

The Toolkit will remind you automatically when an update is required, and the code and any resources for your plugin will be preserved.

Structure of the inspector projects

General

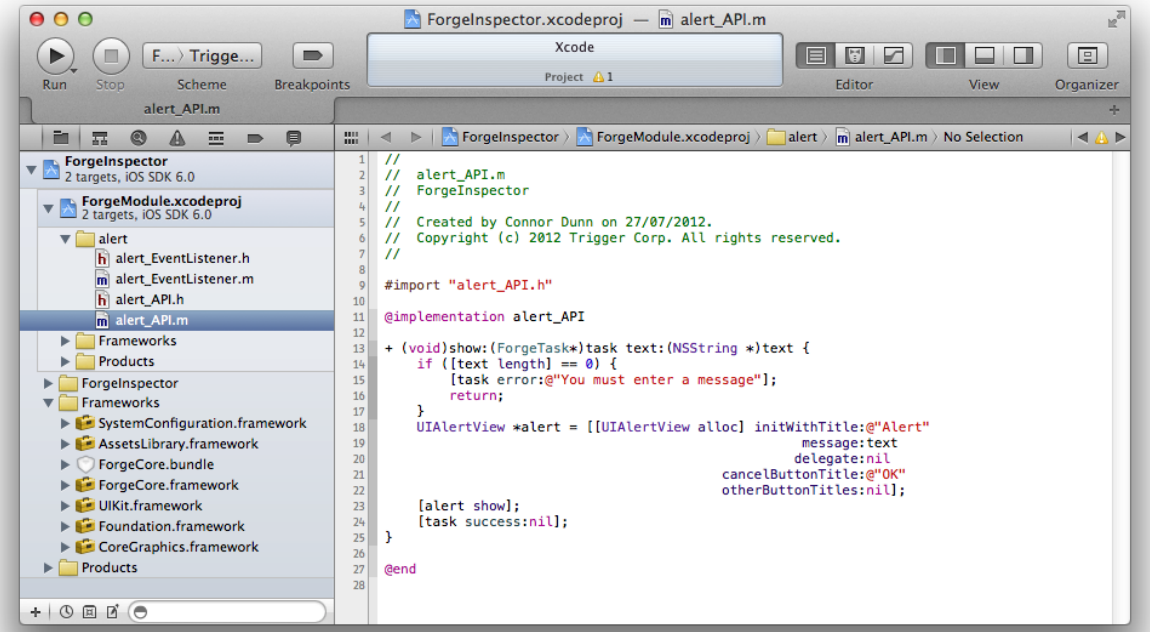
- The assets folder contains files loaded into the webview, i.e. HTML/JS/CSS files.
- The `assets/forge` folder contains a prebuilt `all.js` which should not need to be modified.
- The `assets/src` folder contains a prebuilt Forge app designed for testing plugins, allowing you to run API methods from a simple web based GUI within the app. It is not recommended you modify this app, as it is re-generated and overwritten when the Inspector project is updated. Instead, it is better to write tests for your plugin - see *Testing your plugin*.

Android

- Plugins must contain a package `io.trigger.forge.android.modules.<plugin>` where `<plugin>` is the plugin name you gave at creation time.
- This package **must** contain an API.java file which is the API exposed to Javascript, see API methods.
- It can also contain an EventListener.java to listen for native events, see *Events*.
- An example plugin is included in `io.trigger.forge.android.modules.alert`

Important: On Android you should only need to modify files in `ForgeModule/src` directly, any other changes should be done using build steps.

iOS The iOS inspector project should look something like this:



```
//
//  alert_API.m
//  ForgeInspector
//
//  Created by Connor Dunn on 27/07/2012.
//  Copyright (c) 2012 Trigger Corp. All rights reserved.
//

#import "alert_API.h"

@implementation alert_API

+ (void)show:(ForgeTask*)task text:(NSString *)text {
    if ([text length] == 0) {
        [task error:@"You must enter a message"];
        return;
    }
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alert"
                                                    message:text
                                                   delegate:nil
                                          cancelButtonTitle:@"OK"
                                          otherButtonTitles:nil];
    [alert show];
    [task success:nil];
}

@end
```

The ForgeModule subproject is used to contain the code and resources for your plugin, and the ForgeInspector outer-project is the sandbox you can run and test your plugin in before packaging it up to send to Trigger.io.

- There is no namespacing in Objective-C, you can create files in any structure you like in the ForgeInspector project.
- Files to be included in the plugin build should be in the ForgeModule project and included in the ForgeModule target.
- Plugins **must** include a `<plugin>_API.m` file which is the API exposed to Javascript. See *Exposing native APIs to Javascript*.
- Plugins can also contain `<plugin>_EventListener.m`, to listen for native events, see *Handling native events*.
- An example plugin is included in `ForgeModule/alert/alert_API.m`

Important: On iOS you should only add or change files in the ForgeModule project.

Structure of a plugin In order to upload a plugin you must put the files that make up a plugin, along with a manifest for the plugin in a particular structure in a folder. To help you get started, the Trigger Toolkit can create an initial plugin folder and `manifest.json` for you. To do this, choose "Create new local version" after creating a new plugin in the Toolkit.

Plugins take the following structure:

```
.trigger/                       - Contains code used by the Toolkit to help develop your plu
plugin/                         - The parts of your plugin that are uploaded to be used whe
       manifest.json            - Contains the basic properties for your plugin
       android/                 - Folder containing all android related files
               plugin.jar       - Built Android code
               build_steps.json          - Android build steps, see native build steps
               res/                      - Android resource files, see including resources
                   values/
                          myvalues.xml
               libs/                     - Android libraries
                    mysdk.jar
                    arm/
                        mynativesdk.so
       ios/                              - Folder containing iOS related files
           plugin.a                      - Built iOS plugin
           build_steps.json              - iOS build steps
           bundles/                      - iOS bundles (resources) to include
                   myplugin.bundle
                   mysdk.bundle
       javascript/
                  plugin.js              - Javascript code for your plugin, generally used to expose
       tests/                            - see Testing your plugin
             automated.js                - Automated tests for your plugin
             interactive.js               - Tests for your plugin that require user interaction
             fixtures/                   - Files your tests require to run
                      test.png
inspector/                               - Inspector projects used to develop your plugin
          an-inspector/                  - Android inspector project
          ios-inspector/                 - iOS inspector project
          ios-inspector.2012-11-19       - A backup of a previous version of the iOS inspector
```

manifest.json The manifest for a plugin looks something like:

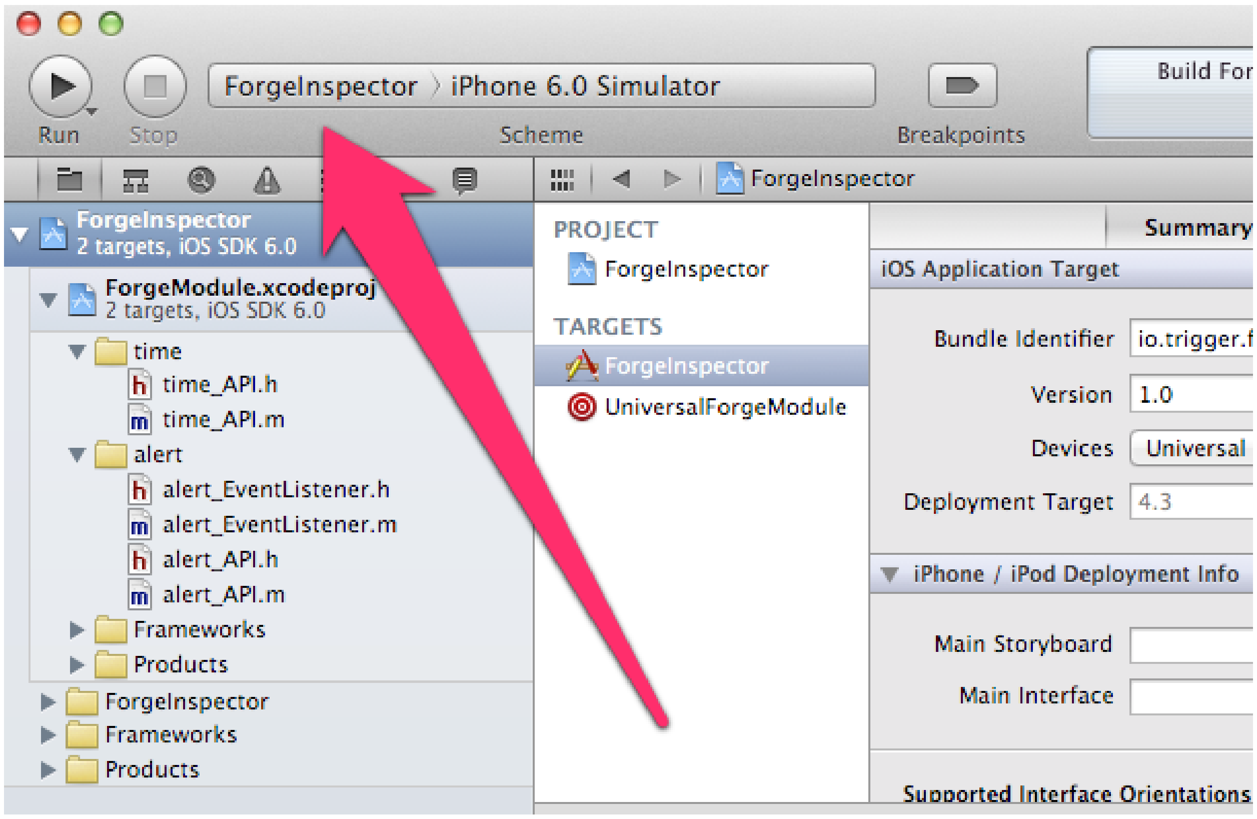
```
{
    "description": "Example alert box plugin",
    "name": "alert",
    "uuid": "e5ed6305192f11f4efde406c8f074dfa",
    "version": "1.0",
    "platform_version": "v1.4.26"
}
```

All of its fields are required - a template manifest.json will be generated for you when you create your plugin in the toolkit.

Note: The platform version for your plugin does not need to match your app, you only need to update your plugins platform version if you require newer plugin feature, or if the Toolkit prompts you to.

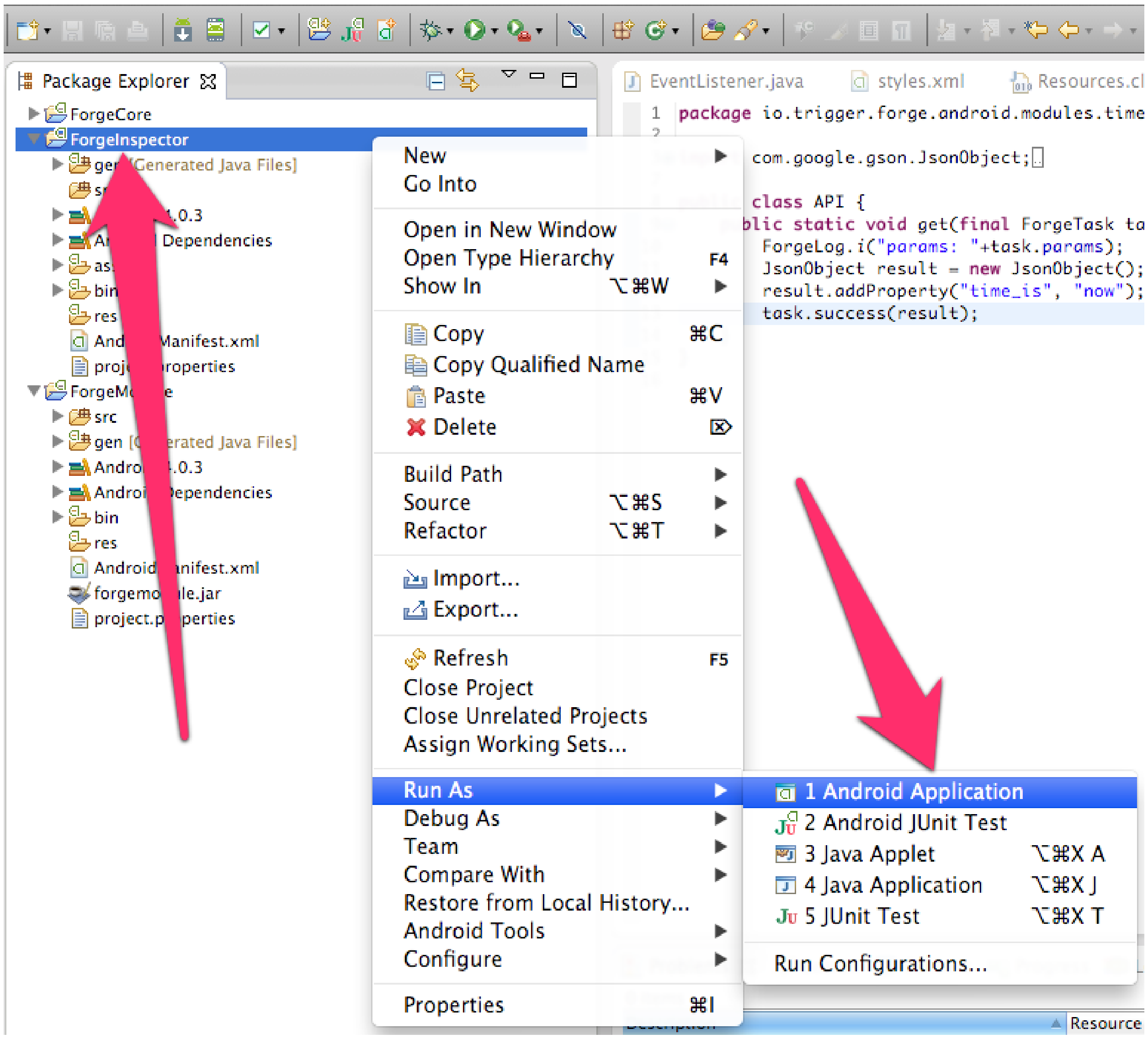
Testing your plugin An inspector app is included which allows you to view and invoke all available API methods, as well as run your test files.

To run the inspector app from Xcode, select the `ForgeInspector` target and an appropriate device or simulator:



Bundle Identifier	io.trigger.f
Version	1.0
Devices	Universal
Deployment Target	4.3

In Eclipse, select the `ForgeInspector` project and choose to run it as an Android application.



```
package io.trigger.forge.android.modules.time

                  com.google.gson.JsonObject;

                  class API {
                 blic static void get(final ForgeTask ta
                   ForgeLog.i("params: "+task.params);
                   JsonObject result = new JsonObject();
                   result.addProperty("time_is", "now");
                   task.success(result);
```

For more detailed information on writing and running tests for your plugin see *Testing your plugin*.

Building/packaging your plugin

Android To build and export your plugin to be included in an actual Forge app:

1. Right click the `src` folder and choose Export...
2. Use the wizard to export the contents of the folder as a JAR
3. `Export generated class files and resources` should be checked.
4. Save that jar as `android/plugin.jar` in your plugin folder.

iOS To build and export your plugin to be included in an actual Forge app, choose the `UniversalForgeModule` target and press Run. A file `build/plugin.a` should appear in the ForgeInspector folder: save that file as `ios/plugin.a` in your plugin folder.

Expected workflow The inspector app is a convenient way to check that your plugin works properly, before exporting it and uploading it to Trigger.io.

Using the default app supplied by the inspector app, you can send messages to your plugin to check it responds correctly, and check that it fires the right Javascript events when required.

To perform more detailed tests of your plugins you should write automated or interactive test files, these will allow you to fully exercise your plugin through the inspector project.

Exposing native APIs to Javascript One use of native plugins is to expose to Javascript a number of API methods that run native code. To facilitate this, Forge provides a structure to simplify communication between Javascript and native code.

Important: API Methods are always asynchronous: this means you provide a callback in Javascript which will be called with the result, and other Javascript code may excute while waiting for the result. This also means in native code you can perform tasks that don't return immediately without blocking Javascript execution.

Javascript To make an API call from Javascript the following code is used:

```
forge.internal.call(
    'alert.show',
    {text: 'Test'},
    function () { alert('Success!') },
    function (e) { alert('Error: '+e.message)}
)
```

The `forge.internal.call` method sends a message to native code, it takes 4 parameters:

- `alert.show` is a string representing the native method which will be called: in this case the method `show` will be called from the plugin `alert`. See the Android and iOS sections below for more detail on how the native method will be found.
- The second parameter is an object containing data to be passed back to the method. Top level properties in this object can be passed directly to parameters in the method (see below for details). This object must be JSON serializable.
- The third parameter is a success callback, which may be called with data returned from native code.
- The forth parameter is an error callback, this may also be called with data, it is conventional to return a human readable description of the error in the property `message` of the returned object.

To expose an API similar to standard Forge modules you can include JavaScript code with your plugin by placing it in `javascript/plugin.js` in your plugin directory. to expose the `alert.show` method above you could use code such as:

```
forge.alert = {
    show: function (text, success, error) {
        forge.internal.call('alert.show', {text: text}, success, error);
    }
};
```

Android To expose API methods in Android they must be in API.java in the package `io.trigger.forge.android.modules.<plugin>` where `<plugin>` is your plugin name.

To explain the structure of an Android API method we can look at the `alert.show` method included in the inspector project:

```
public static void show(final ForgeTask task, @ForgeParam("text") final String text) {
    if (text.length() == 0) {
        // Error if there is no text to show
        task.error("No text entered");
        return;
    }
    AlertDialog.Builder builder = new AlertDialog.Builder(ForgeApp.getActivity());
    builder.setMessage(text).setCancelable(false).setPositiveButton("Ok",
        new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                task.success();
            }
        }
    );
    AlertDialog alert = builder.create();
    alert.show();
}
```

Firstly, looking at the method signature:

- All exposed API methods should be `public static void` methods.
- The first parameter is always a

  object, which contains information about the task as well as helper methods to complete the task and to return a response.
- Additional parameters must be `String`, `long`, `int`, `double`, `JsonArray` or `JsonObject` and must have a `@ForgeParam` annotation.
- annotations pass properties with the given name from Javascript directly to parameters in the API method, after checking they exist and are the right type.
- Parameters do not need to be specified and the full passed from Javascript can be accessed through `task.params`.

Most of the method body is code to display the alert dialog in Android, the important lines to notice related to Forge are:

- `task.error("No text entered");` - Returns the string given to the error callback in Javascript.
- `task.success();` - Calls the success callback in Javascript with no arguments.
- We can see the success method is only called when the alert dialog button is clicked, which means it happens asynchronously: this is not a problem.

All API methods should call `task.error()` or `task.success()` **exactly once**: if a method needs to return values to Javascript multiple times then events should be used.

iOS API methods are exposed in iOS by creating a class called `<plugin>_API` within the ForgeModule project where `<plugin>` is your plugin name.

The structure of an API method can be seen in the example included in the inspector project:

```
+ (void)show:(ForgeTask*)task text:(NSString *)text {
    if ([text length] == 0) {
        [task error:@"You must enter a message"];
        return;
    }
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Alert"
                                                     message:text
```

```
                                                             delegate:nil
                                                    cancelButtonTitle:@"OK"
                                                    otherButtonTitles:nil];
    [alert show];
    [task success:nil];
}
```

The method signature defines the API method:

- All exposed API methods are `+ (void)` methods.
- The name of the exposed method is taken up to the first `:`, so in this case is `show`
- The first parameter to API methods is a

  object, which contains information about the task as well as helper methods to complete the task and to return a response.
- Additional parameters must be `NSString`, `NSNumber`, `NSDictionary` or `NSArray`, the name of the parameter will be used to extract the argument from the javascript parameters object. Type checking is not performed on iOS.
- Any parameters not specified in the signature can be accessed through `task.params`

The method body contains the following Forge specific features:

- `[task error:@"You must enter a message"];` - Returns a string to the error callback in Javascript
- `[task success:nil];` - Returns no parameters to the success callback in Javascript

All API methods should call `[task error:]` or `[task success:]` exactly once.

Triggering Javascript events There are two distinct types of events in plugins:

- **Javascript events** which can be triggered from native code at any point, used for situations that aren't a direct response to an API method call.
- **Native events** which are points in an application's execution that plugins can hook into and execute their own code. For example plugins can execute native code on application start, without having to use any Javascript.

Quite frequently you'll want to handle a native event and trigger and trigger a corresponding Javascript event. This section of the plugin docs will discuss the second type of event, those which are triggered by the device. If you're interested in the second type of event described, see *Handling native events*.

Javascript An example of listening for a Javascript event is included in the inspector project:

```
forge.internal.addEventListener("alert.resume", function () {
    alert("Welcome back!");
});
```

- Events are identified by a unique string, it is conventional to use the format `plugin.eventName` for this string as shown above.
- The second parameter is a callback to be invoked whenever the event is triggered from native code: data can be passed in the first parameter to this callback.

Android To trigger a Javascript event from native Android code, the

method can be called at any point. For example:

```
ForgeApp.event("alert.resume", null);
```

- The first parameter is the event identification string which is listened for in Javascript.
- The second is an optional object to pass back to Javascript.

iOS To trigger Javascript events from native iOS code, use `[ForgeApp event]` at any point. For example:

```
[[ForgeApp sharedApp] event:@"alert.resume" withParam:nil];
```

- The first parameter is the event identification string which is listened for in Javascript.
- The second is an optional object to pass back to Javascript.

Handling native events There are two distinct types of events in plugins:

- **Javascript events** which can be triggered from native code at any point, used for situations that aren't a direct response to an API method call.
- **Native events** which are points in an application's execution that plugins can hook into and execute their own code. For example plugins can execute native code on application start, without having to use any Javascript.

Quite frequently you'll want to handle a native event and trigger a corresponding Javascript event. This section of the plugin docs will discuss the second type of event, those which are triggered by the device. If you're interested in the first type of event described, see *Triggering Javascript events*.

Android To listen for native events in Android an `EventListener` class must be added to your plugin's package. For example, `io.trigger.forge.android.modules.alert.EventListener`. This class should extend the `ForgeEventListener` class and implement any of the methods it wants to listen for. See the for the list of methods available for override as well as their meaning.

The example EventListener from the inspector project looks like:

```
public class EventListener extends ForgeEventListener {
    @Override
    public void onRestart() {
        ForgeApp.event("alert.resume", null);
    }
}
```

- `@Override` on the method ensures the method exists in ForgeEventListener and is an available event.
- If you want to prevent subsequent event listeners from receiving an event, you should return a non-`null` value. For example, if a key press event is handled by a plugin it can return `true` and prevent other plugins from seeing the event.

iOS To listen for native events in iOS, a class called `<plugin>_EventListener` must be created where `<plugin>` is your plugin name. This class should extend `ForgeEventListener` and implement any of the methods it wants to listen for. See the for the list of methods available for override as well as their meaning.

The example EventListener from the inspector project looks like:

```
@interface alert_EventListener : ForgeEventListener

@end

@implementation alert_EventListener

+ (void)applicationWillEnterForeground:(UIApplication *)application {
    [[ForgeApp sharedApp] event:@"alert.resume" withParam:nil];
}

@end
```

- If you want to prevent subsequent event listeners from receiving an event, you should return a non-`nil` value. For example, if a key press event is handled by a plugin it can return `YES` and prevent other plugins from seeing the event.

Communicating with native code When making native API calls or triggering events it is useful to send more than just a string as data between the JavaScript and native code (in both directions).

On both Android and iOS you are able to send either simple pieces of data (numbers or strings) or JavaScript style objects and arrays in both directions. In JavaScript this means you can send any JSON serializable data to native, and you will always recieve back JSON parsed variables.

Android To encode and decode this JSON data in Java we use the Gson library. This provides a fast way of working with JSON with a simple and clear API. When working on Android you should keep the following in mind:

- When returning data to JavaScript it should be a .
  - Simple types can be converted to a `JsonElement` using the constructor: `new JsonPrimitive("My string")` or `new JsonPrimitive(50)`.
  - More complex types can be created using and .
  - `ForgeTask.success` and `ForgeApp.event` will also directly accept a `String` and create the `JsonElement` for you.
- `ForgeTask.params` is of type . To read the string `test` from `{"test":"My Data"}` sent from Javascript the following code could be used `task.params.get("test").getAsString()`.
- If accepting parameters directly using the annotation then the types `JsonObject` and `JsonArray` must be used rather than `JSONObject` and `JSONArray`.
- `null` passed to and from JavaScript is represented by `JsonNull.INSTANCE`.

iOS On iOS we use JSONKit to encode and decode JSON data. When working on iOS you should keep the following in mind:

- JSONKit maps JSON on to the familiar NSNull, NSString, NSNumber, NSArray and NSDictionary objects: any time you return data to JavaScript you should return one of these types. Any time you receive data from Javascript, you should expect one of these types.
- JavaScript booleans are encoded as NSNumber using `[NSNumber numberWithBool:YES/NO]`

Using external libraries Often it is useful to include external libraries in your plugin, this can be done on both Android and iOS.

Important: Class names must be unique across all plugins and the core Forge app: this means multiple plugins cannot include the same libraries. In the future we will be supporting dependencies between plugins to allow shared libaries; until then if multiple plugins include the same libraries only one of them can be enabled for any individual app.

Android To include libraries copy them into the `plugin/android/libs` folder in your plugin. Once in place update your inspector project through the Toolkit and the libraries will be included in the inspector project as they would be in a Forge build.

Both Java and native libraries are supported: native libraries should be placed in a sub-folder to indicate the architecture they're built for - see *Structure of a plugin*.

If the library you want to include is distributed as unbuilt Java files, you can include that code in your own plugin source tree and export it as part of the JAR file from Eclipse.

iOS On iOS, there are two two types of external libraries: system frameworks made available by Apple, and 3rd party frameworks.

System frameworks To include Apple frameworks an *add_ios_system_framework build step* must be added to link with the framework at build time.

Once added the inspector project should be updated to apply this change while you develop your plugin.

3rd party libraries/frameworks There are a number of different formats used to distribute 3rd party libraries for iOS: not all of them are directly compatibile with Forge, but with some minor changes they can all be included in a plugin:

- If the library is a .a file, simply include it in the ForgeModule project and add it to the ForgeModule target. If the library includes separate header files you can also add these to the ForgeModule project.
- If the library is a framework you will need to browse the contents of the framework and add the main framework file (which will be the name of the framework with no file extension) and any headers to the ForgeModule manually.
- If the framework contains any .bundle files you will need to include them in the plugin (see *Including resources*).

Changing build configuration Plugins often need to change properties of the app which cannot be set at runtime, such as app permissions on Android, or linked system frameworks on iOS. These changes can be described in the native build steps file for each platform, and will be applied at build time for the Forge app.

The build steps go in either `android/build_steps.json` or `ios/build_steps.json` in the plugin's folder. These JSON files take the following format:

```
[
    {
        "do": {
            "build_task": {
                "parameter": "value"
            }
        }
    }, {
        "do": {
            "second_build_task": {
                "input": "data",
                "other_input": "data2"
            }
        }
    }
]
```

This consists of an array of tasks to perform before the build is completed.

The types of task that can be performed and the parameters that need to be passed to each task varies by platform and is described below.

Note: After changing the build steps for either Android or iOS it is important to update the inspector project, any new build steps will be applied to the project to keep your development environment as close as possible to the final Forge build.

General

`include_dependencies` This build step can contain a dictionary of dependencies and their details, each dependency must contain a `hash`, for example:

```
{
    "do": {
        "include_dependencies": {
            "my_library": {
                "hash": "01230123012301230123012301230123"
            },
            "my_other_library": {
                "hash": "45674567456745674567456745674567"
            }
        }
    }
}
```

In the future these dependencies will be selectable via the Toolkit, until then a list of currently available dependencies can be found at: *native_plugins_shared_dependencies*.

Android

`android_add_permission`

- `permission`: the permission to add, i.e. `android.permission.CAMERA`

Example:

```
{
    "do": {
        "android_add_permission": {
            "permission": "android.permission.CAMERA"
        }
    }
}
```

`android_add_feature`

- `feature`: the feature to request
- `required`: Whether or not the feature is required, `"true"` or `"false"`

Example:

```
{
    "do": {
        "android_add_feature": {
            "feature": "android.hardware.camera",
            "required": "true"
        }
    }
}
```

`android_add_activity`

- `activity_name`: Name of activity
- `attributes`: Optional attributes

Example:

```
{
    "do": {
        "android_add_activity": {
            "activity_name": "com.example.sdk.MyActivity",
            "attributes": {
                "android:screenOrientation": "portrait"
            }
        }
    }
}
```

`android_add_service`

- `service_name`: Name of service
- `attributes`: Optional attributes

Example:

```
{
    "do": {
        "android_add_service": {
            "service_name": "com.example.sdk.MyService"
        }
    }
}
```

`android_add_receiver`

- `receiver_name`: Name of receiver
- `attributes`: Optional attributes
- `intent_filters`: Optional intent filters

Example:

```
{
    "do": {
        "android_add_receiver": {
            "receiver_name": "com.example.sdk.MyReceiver",
            "intent_filters": [{
                "action": "android.intent.action.BOOT_COMPLETED"
            }]
        }
    }
}
```

iOS

add_ios_system_framework

- `framework`: the framework to add

Example:

```
{
    "do": {
        "add_ios_system_framework": {
            "framework": "CoreMedia.framework"
        }
    }
}
```

ios_add_url_handler

- `scheme`: URL scheme to handle

Example:

```
{
    "do": {
        "ios_add_url_handler": {
            "scheme": "myurlscheme"
        }
    }
}
```

set_in_info_plist

- `key`: Key to add/change: you can use `a.b` to change key `b` nested inside `a`
- `value`: Value to set it to

Example:

```
{
    "do": {
        "set_in_info_plist": {
            "key": "MyKey",
            "value": "My Data"
        }
        }
}
```

Including resources Native code sometimes requires additional resources, such as layout or image files. These work in slightly different ways on Android and iOS but can be easily included in plugins for both.

Android To include resources they must be placed in the `plugin/android/res` folder, once there the inspector project can be updated and the resource files will be included.

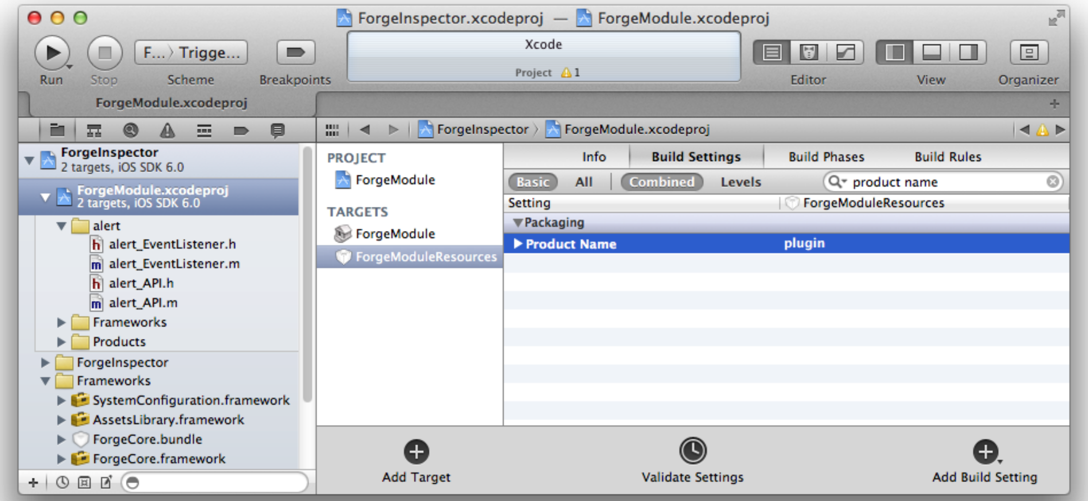
For more information about what kind of resources can be added to your plugin and how they can be used with the Android SDK, see the official Android docs concerning App Resources.

iOS On iOS, any resources included in your plugin must be part of a bundle.

3rd party bundles can be included by placing the bundles in `plugin/ios/bundles`. An inspector project update will then be required to include them in the inspector for development

If you have resources such as images that you want to include in your plugin, you must create your own bundle. The inspector project includes a `ForgeModuleResources` target which can be used to help with this:

1. All bundles must have a unique name - you can name your bundle by setting the `Product Name` for the ForgeModuleResources target.



Product Name plugin

2. Add the required resources to the `ForgeModule` project and include them in the `ForgeModuleResources` target.

3. Use the named bundle to reference these resources from your native code, with code like this:

   ```
   NSString * bundlePath = [[NSBundle mainBundle] pathForResource:@"my_bundle_name" ofType:@"bundle
   NSBundle * myBundle = [NSBundle bundleWithPath:bundlePath];
   ```

   For more information, see Accessing a Bundle's Contents.

4. When building `UniversalForgeModule` your bundle will be output in the build folder: make sure to include this in `ios/bundles/` in your plugin folder.

5. After adding the bundle to `ios/bundles`, you'll need to re-generate your Inspector before continuing.

Working with files Forge plugins will often want to consume or produce files (or both). To make this easy Forge has a ForgeFile object which can be passed efficiently between plugins and between native and JavaScript code.

Javascript ForgeFile objects can be represented as simple JavaScript objects. These objects must contain a `uri` which references an actual file that either Forge or a loaded plugin will be able to resolve at a later point. They should also contain a `name` and a `mimeType` if sensible values for these are known. Some file objects will contain additional properties such as `width` and `height` which may be used when processing the file.

The `file` module has a number of methods which accept file objects.

For information on testing with files see *Testing your plugin*.

Android

Receiving from JavaScript When receiving a file from JavaScript it will be of type JsonObject, this can be passed into the ForgeFile constructor to access various helper fuctions for the file. For example:

```
ForgeFile file = new ForgeFile(ForgeApp.getActivity(), task.params.get("file"));
byte[] fileData = null;
try {
    fileData = file.data();
} catch (IOException e) {
    // handle properly
}
```

Further details on the ForgeFile class in Android can be found in the API docs:

Returning to JavaScript To return a file to JavaScript (which can then be used with other modules, such as forge.request.ajax) a JsonObject with the appropriate properties must be constructed and returned. For example:

```
JsonObject file = new JsonObject();
file.addProperty("uri", "file:///data/path/to/file");
task.success(file);
```

iOS

Receiving from JavaScript When receiving a file from JavaScript it will be of type NSDictionary, this can be passed into the ForgeFile constructor to access various helper fuctions for the file. For example:

```
ForgeFile* file = [[ForgeFile alloc] initWithFile:[task.params objectForKey:@"file"]];
NSString* fileURL = [file url];
```

Further details on the ForgeFile class in iOS can be found in the API docs:

Returning to JavaScript To return a file to JavaScript (which can then be used with other modules, such as forge.request.ajax) an NSDictionary with the appropriate properties must be constructed and returned. For example:

```
NSDictionary* file = @{@"uri": @"/path/to/file"};
[task success:file];
```

Testing your plugin As well as allowing you to directly call exposed API methods, the Inspector project will run tests you include with your plugin. These tests are written in JavaScript, using QUnit, and are split into automated and interactive sections. To run the tests simply press the appropriate button when running the Inspector project.

Automated tests To include automated tests you should create a file `tests/automated.js` in your `plugin` folder: it is important that these tests complete without requiring user input as we may run them automatically to check compatibility with new platform versions. Test as much of your API as possible in these tests, so that we can give you early warning about incompatibility with new platform versions and other plugins.

An example automated test (from the `prefs` module) could be:

```
asyncTest("Set and get a pref (Number)", 1, function() {
    var pref = "test"+Math.random();
    var value = Math.random();
    forge.prefs.set(pref, value, function () {
        forge.prefs.get(pref, function (newValue) {
            equal(newValue, value, "Preference value which was set");
            start();
        });
    });
});
```

Further documentation on available QUnit methods is available at: http://api.qunitjs.com/.

Interactive tests Sometimes tests require user interaction - in this case you should include them in `tests/interactive.js` in your `plugin` folder. These tests will never be run automatically, but placing them here is a good way for you to be able to test the functionality of your plugin as you develop it.

A helper function `askQuestion(question, answers)` is provided to make it easier to prompt the user for input: `question` is the question to ask, and `answers` is a mapping of answers to callback functions.

An example interactive test (this time taken from the `barcode` module) could be:

```
asyncTest("Scan barcode", 1, function() {
    askQuestion("Does this device have a camera? If yes when prompted scan a barcode", {
        Yes: function () {
            forge.barcode.scanWithFormat(function (barcode) {
                askQuestion("Is this your barcode: "+barcode.value+" and was it a: "+barcode.format,
                    Yes: function () {
                        ok(true, "User claims success");
                        start();
                    },
                    No: function () {
                        ok(false, "User claims failure");
                        start();
                    }
                });
            }, function (e) {
                ok(false, "API call failure: "+e.message);
                start();
            });
        },
        No: function () {
            ok(true, "No camera");
            start();
        }
    });
});
```

Fixtures If you need to use additional resources (such as an image file) as part of your test, you can place them in the `tests/fixtures` folder in your plugin. These files will be included in `src/fixtures/<plugin name>/` when you update the Inspector app.

If your plugin has an API which accepts a ForgeFile object as described in *Working with files*, it can be useful to create file objects to test with. Calling `forge.inspector.getFixture("plugin", "file.png")` will return a ForgeFile object for the fixture "file.png".

API docs Documentation in Javadoc and appledoc format is available for the classes you can use in your plugins on Android and iOS

Common problems This page details some of the common problems when developing plugins. If you can't find information on the problem you are having then you can email support@trigger.io or check http://stackoverflow.com/questions/tagged/trigger.io.

Toolkit error messages As plugins have a greater affect on the build process it is far easier for a plugin to cause issues when trying to build an app using it, to limit this the Toolkit will try to detect common problems while you develop your plugin. This page contains some more detailed information about the various errors that can appear while developing, and how you can fix them.

iOS/Android inspector not found This is shown when your plugin contains an Android or iOS folder but no matching inspector project is found.

iOS/Android inspector out of date This message is shown when your current inspector project was not created with the files included in your plugins folder and with the current plugins platform version. This generally indicates you should update your inspector project before continuing development.

The directory 'x' was expected to be a file This error is shown if a directory is found in a location that a file was expected, for example if a plugins jar file was actually an extracted folder rather than a compressed file.

The file 'x' was expected to be a directory Similarly this error is shown if a file is found in a location that was expected to be a directory. An example of this would be if an iOS bundle was a file, iOS bundles are actually a special type of directory that appears as a single file in Finder.

The path 'x' is required and was not found This error means a file that is required based on the current structure of your plugin was not found. The most common example of this is if an `android` folder is found within your plugin then a `plugin.jar` file must be included within that folder.

The path 'x' was found but not expected This warning means a file or directory was found with a path that was not expected, this generally won't cause a problem in itself, but this file will not be used when including the plugin in a Forge build, and could mean it is named incorrectly or placed in the wrong location.

File 'android/plugin.jar' is not a valid jar file containing a Forge Android plugin with name 'x' The Toolkit checks the jar file for your plugin contains a class which looks like it belongs to your plugin, if it can't find it then something is probably wrong with the structure of your code or with the way you created you jar file. Check your plugin is named correctly and you include a API.java for that module name in your jar, if you still see problems then get in touch so we can look into it.

File 'ios/plugin.a' is not a valid static library file containing a Forge iOS plugin with name 'x' The Toolkit checks your built iOS plugin contains classes that look like they belong to your plugin, if it can't find them then something is probably wrong with the file you've built. Read through the docs and rebuild your plugin, if you can't solve the problem get in touch and we'll be able to help you out.

Validation for 'x' failed: ... If your error message isn't explictly listed above but starts with this text it means the given file failed some kind of pre-upload validation we perform, often a more descriptive error message will be given. Check the file follows this documentation and if you still can't resolve the issue get in touch.

Browser add-on

`activations`: Inject files into pages Platforms: Browser

The `activations` module allows styles and scripts to be injected into webpages with specified URLs.

Config The `activations` module must be enabled in config.json as follows:

```
{
    "modules": {
        "activations": [
            {
                "patterns": ["http://mail.google.com"],
                "scripts": ["gmail.js"],
                "styles": ["gmail.css"],
                "run_at": "start",
                "all_frames": false
            }
        ]
    }
}
```

This field specifies when and how your foreground files will be embedded into pages. It is an array of objects with three required keys:

- `patterns` is an array of Match Patterns which control on which URLs your app will activate
- `scripts` is an array of Javascript files which will be embedded
- `styles` is an array of CSS files which will be embedded

As well as an optional keys:

- `run_at` optionally defines when your included scripts will be added to the page, must be one of the following:
- `"start"` scripts will be run immediately, potentially before the DOM is ready
- `"ready"` scripts will run as soon as the DOM is ready
- `"end"` (default) scripts will run at some point after the DOM is ready, with no guarantees as to whether or not `window.onload` will have fired yet or not.
- `all_frames` optionally defines whether activations will be run in all frames or just the top level document, by default it is false.

Important: Safari only supports a single object in the activations array.

`background`: Background scripts Platforms: Browser

The `background` module allows your app to have a persistent Javascript context running in the background

Config Browsers have the *concept of content scripts and background* files.

This field lists the files that should be included in background context.

```
{
    "modules": {
        "background": {
            "files": ["js/background.js"]
        }
    }
}
```

`button`: Toolbar button Platforms: Browser

Config The `button` configuration controls the appearance and function of toolbar icons in the browsers. With this directive, you can specify a HTML file which will be displayed when the button is clicked, a default button icon as well as platform-specific icons.

```
{
    "modules": {
        "button": {
            "default_popup": "popup.html",
            "default_title": "Button Title",
            "default_icon": "my-default-icon.ico",
            "default_icons": {
                "firefox": "my-icon-for-firefox.ico"
            }
        }
    }
}
```

- `default_popup` should refer to a local HTML file, included in your app, which will be displayed after the button is clicked; for more information, see *part I of the tutorial*
- `default_title` this will show as the tooltip text when users hover over the toolbar button. It is a required field for IE.
- `default_icon` should refer to a local image file, included in your app, to be used as the button icon
- `default_icons` allows you to override the `default_icon` icon, one platform at a time: the object keys should be one or more of `chrome`, `firefox`, `safari` or `ie`

Important: Internet Explorer requires you to override the `default_icon` with a 16x16 bitmap in .ico format.

API Adding and manipulating a toolbar button near the browser's address bar.

`setIcon` Platforms: Browser only

Sets the icon for the toolbar button.

`button.`**`setIcon`**(*url*, *success*, *error*)

Arguments

- **url** (*string*) – the URL of the icon
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setURL` **Platforms: Browser only**

Sets the path to the HTML page that should be opened when the toolbar button is clicked.

`button.`**`setUrl`**`(`*url*, *success*, *error*`)`

Arguments

- **url** (*string*) – relative URL to the HTML to set as the toolbar button popup
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`onClicked.addListener` **Platforms: Browser only**

Sets a function to be executed when the toolbar button is clicked.

`button.onClicked.`**`addListener`**`(`*callback*`)`

Arguments

- **callback** (*function()*) – function to be executed when the toolbar button is clicked.

`setBadge` **Platforms: Browser only (Not supported on Internet Explorer)**

Sets a number to appear as a notification badge on the toolbar button.

`button.`**`setBadge`**`(`*number*, *success*, *error*`)`

Arguments

- **number** (*number*) – number to display as badge
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setBadgeBackgroundColor` **Platforms: Browser only (Not supported on Safari or Internet Explorer)**

Sets the background color for the badge.

`button.`**`setBadgeBackgroundColor`**`(`*color*, *success*, *error*`)`

Arguments

- **color** (*array*) – an array of four integers in the range [0,255] that make up the RGBA color of the badge. For example, opaque red is [255, 0, 0, 255].
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`setTitle` Platforms: Browser only

Set the tooltip text for a toolbar button.

`button.setTitle(`*title*, *success*, *error*`)`

Arguments

- **title** (*string*) – title text to set as the toolbar tooltip
- **success** (*function()*) – callback to be invoked when no errors occur
- **error** (*function(content)*) – called with details of any error which may occur

`document`: Document utility functions Internet Explorer features a number of inconsistencies with some common document operations, these methods allow you to work around these problems in a platform-independent manner.

API

`reload` Platforms: Browser Only

`document.reload()`

Internet Explorer's implementation of document.reload() only refreshes the static content of a page. This method will reload a page and force all scripts to be re-evaluated as well.

`location` Platforms: All

`document.location(`*success*, *error*`)`

Arguments

- **success** (*function(location)*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

Internet Explorer versions 9 and later aborts page loading with a permission denied message if the document location is accessed from within an iframe. This function provides a workaround which functions correctly under all browsers.

Error object properties:

- `statusCode`: Status code returned from the server.
- `content`: Content returned from the server (if available).

Example:

```
forge.document.location(function(location) {
    forge.logging.log(location.href);
  }, function(error) {
    alert('Failed to get document location: '+error.message);
  }
});
```

`icons`: App icons This part of the config allows you to define the icons to be used for your app. All icons are square, and must be placed in your `src` directory.

Define your desired icons with `"size": "path"` attributes, where `size` is the pixel height (and width) of the icon, and `path` is where the image has been placed under the `src` directory.

You can specify different icons for different platforms as so:

```
"android": {
    "36": "icon36.png",
    "48": "icon48-android.png",
    "72": "icon72.png"
},
"chrome": {
    "16": "icon16.png",
    "48": "icon48-chrome.png",
    "128": "icon128.png"
}
```

Here, Android and Chrome will share their 16x16 pixel icon, but use different 48x48 pixel icons.

The icons required for each platform are listed below:

- Android: 36px, 48px and 72px
- Chrome: 16px, 48px and 128px
- Firefox: 32px and 64px
- Internet Explorer: 16px `.ico` format
- iOS: 57px, 72px, 114px and 144px for home screen icons, 512px to be shown in iTunes.
- Safari: 32px, 48px and 64px with transparent background. See the Creating an Image section in Apple's Safari extension guide.

Note: If you specify *any* icons for a particular platform, you **must** specify all required icons!

Note: iOS includes a special `prerendered` option, setting this to true will stop iOS from applying the gloss effect to your icons.

Config

```
{
    "modules": {
        "icons": {
            "android": {
                "36": "icon36.png",
                "48": "icon48-android.png",
                "72": "icon72.png"
            },
            "ios": {
                "57": "icon57.png",
                "72": "icon72-ios.png",
                "114": "icon114.png",
                "144": "icon144.png",
                "prerendered": true
            }
        }
    }
}
```

`is`: Platform Detection Forge allows you to build cross-platform mobile apps and browser extensions from the same code. Sometimes it may be necessary to do a specific action based on the platform that is running the code. The following methods allow you to determine what platform that is.

Config The `is` module must be enabled in `config.json`

```
{
    "modules": {
        "is": true
    }
}
```

API

API

`mobile`

`is.mobile()`

> **Return boolean** Returns true if running on a mobile device

`desktop`

`is.desktop()`

> **Return boolean** Returns true if running on a desktop/laptop computer

`web`

`is.web()`

> **Return boolean** Returns true if running on as a hosted web app

`android`

`is.android()`

> **Return boolean** Returns true if running on an Android device

`ios`

`is.ios()`

> **Return boolean** Returns true if running on an IOS device

`chrome`

`is.chrome()`

> **Return boolean** Returns true if running on Chrome browser

`firefox`

`is.firefox()`

> **Return boolean** Returns true if running on Firefox browser

`safari`

`is.safari()`

> **Return boolean** Returns true if running on Safari browser

`ie`

`is.ie()`

Return boolean Returns true if running on IE browser

`orientation` Platforms: Mobile

`portrait`

`is.orientation.portrait()`

Return boolean Returns true if a mobile device has a portrait orientation

`landscape`

`is.orientation.landscape()`

Return boolean Returns true if a mobile device has a landscape orientation

`connection` Platforms: Mobile

Note: These functions are not reliable during your app's initialisation: you should use *connectionStateChange.addListener*. We guarantee to fire that event as the app starts up.

`connected`

`is.connection.connected()`

Return boolean Returns true if a mobile device has an active internet connection.

`wifi`

`is.connection.wifi()`

Return boolean Returns true if a mobile device is connected via wifi.

`logging`: Logging

Allows you to log a message, and optionally an exception, to the console service provided by the underlying platform.

Config

The `logging.level` configuration directive controls the verbosity of the logging system. It should be set to one of `DEBUG`, `INFO`, `WARNING`, `ERROR` or `CRITICAL`. A setting of `DEBUG` means that all messages will be logged, whereas a setting of `CRITICAL` means that only messages of level `CRITICAL` will be logged.

```
{
    "modules": {
        "logging": {
            "level": ["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"]
        },
    }
}
```

Platforms: Internet Explorer only

The `logging.console` configuration directive enables a log window for the add-on's background page.

```
"logging": {
    "console": true
},
```

API

`log` Platforms: All

`logging.`**`log`**`(`*message[, exception][, level]*`)`

Arguments

- **message** – a string to log on the browser console
- **exception** – a JavaScript Error instance: details from the exception will be appended to your log message
- **level** – importance of this message: one of `forge.logging.DEBUG`, `forge.logging.INFO`, `forge.logging.WARNING`, `forge.logging.ERROR` or `forge.logging.CRITICAL`

`logging.`**`debug`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.DEBUG)`

`logging.`**`info`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.INFO)`

`logging.`**`warning`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.WARNING)`

`logging.`**`error`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.ERROR)`

`logging.`**`critical`**`(`*message*[, *exception*]`)`
Shorthand for `forge.logging.log(message[, exception], forge.logging.CRITICAL)`

`message`: Component communication Platforms: Browser only

It is often useful to be able to send and receive messages between components of your extension. To achieve this in a cross-browser manner, use these methods to broadcast and listen for messages.

Config

The `message` module must be enabled in `config.json`

```
{
    "modules": {
        "message": true
    }
}
```

API

`listen` Platforms: Browser only

Sets up a handler function which will receive messages sent ("broadcast") by your extension.

By supplying the optional *type* parameter, you can filter the messages on which your callback will be invoked; see the *type* parameter to the *broadcast* and *broadcastBackground* methods. If the *type* parameter is omitted, your callback will be invoked for all broadcast messages.

The *callback* parameter will be invoked when a message is to be delivered, with the message contents as its first parameter and a "reply function" as its second parameter. The "reply function" is used to send responses back to the code which broadcast the original message.

`message.listen`([*type*], *callback*, *error*)

Arguments

- **type** (*string*) – (optional) if included, the callback will only be fired for messages broadcast with the same type; if omitted, the callback will be fired for all messages
- **callback** (*function(content,reply)*) – will be called with the contents of relevant broadcast messages as its first parameter and a reply function as its second parameter
- **error** (*function(content)*) – called with details of any error which may occur

`broadcast` Platforms: Browser only

Sends a message to be received by other components of your extension. Messaging will not be recieved by the background page.

The *type* parameter can be used to indicate the purpose of the message and limit the active listeners which will receive the message.

The *callback* parameter will be invoked with any responses from listeners; it may be called multiple times, depending on whether your listeners use the "reply function".

`message.broadcast`(*type*, *content*, *callback*, *error*)

Arguments

- **type** (*string*) – limits the listeners which will receive this message
- **content** (*any*) – the message body
- **callback** (*function(content)*) – invoked each time a listener returns a response to the broadcaster, with the response as its only argument
- **error** (*function(content)*) – called with details of any error which may occur

`broadcastBackground` Platforms: Browser only

Restrictions: Not available from the background page

Sends a message to be received by listeners in your background code: similar to *broadcast*, except that listeners created in individual browser pages will not receive this message; only listeners created in your background page (see *extension-concept-background*).

`message.broadcastBackground`(*type*, *content*, *callback*, *error*)

Arguments

- **type** (*string*) – limits the listeners which will receive this message
- **content** (*any*) – the message body

- **callback** (*function(content)*) – invoked each time a listener returns a response to the broadcaster, with the response as its only argument
- **error** (*function(content)*) – called with details of any error which may occur

`toFocussed` Platforms: Browser only

Like *broadcast*, this method sends a message to be received by content script listeners.

However, not all listeners are passed the message: only the currently focused tab's listeners will receive this message. If the currently focussed tab is not displaying a page your add-on has activated on, no listeners will receive this message.

The callback may be invoked a number of times if several message listeners have been set up per page.

`message.toFocussed`(*type*, *content*, *callback*, *error*)

Arguments

- **type** (*string*) – limits the listeners which will receive this message
- **content** (*any*) – the message body
- **callback** (*function(content)*) – invoked each time a listener returns a response to the broadcaster, with the response as its only argument
- **error** (*function(content)*) – called with details of any error which may occur

Permissions On Chrome this module will add the `tabs` permission to your app, users will be prompted to accept this when they install your app.

`notification`: Notifications

Config The `notification` module must be enabled in `config.json`

```
{
    "modules": {
        "notification": true
    }
}
```

API Notifications allow you to send alerts.

`create` Platforms: All

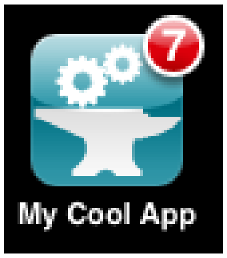
`notification.create`(*title*, *text*, *success*, *error*)

Arguments

- **title** (*string*) – title
- **text** (*string*) – notification message
- **success** (*function()*) – success
- **error** (*function(content)*) – called with details of any error which may occur

`setBadgeNumber` Platforms: iOS

Allows you to set or remove a badge for your app's icon on the iOS home screen:



If you pass in *0* as number, it will remove this badge. This is particularly useful if you want to clear a badge set by a push notification.

`notification.`**`setBadgeNumber`**(*number*, *success*, *error*)

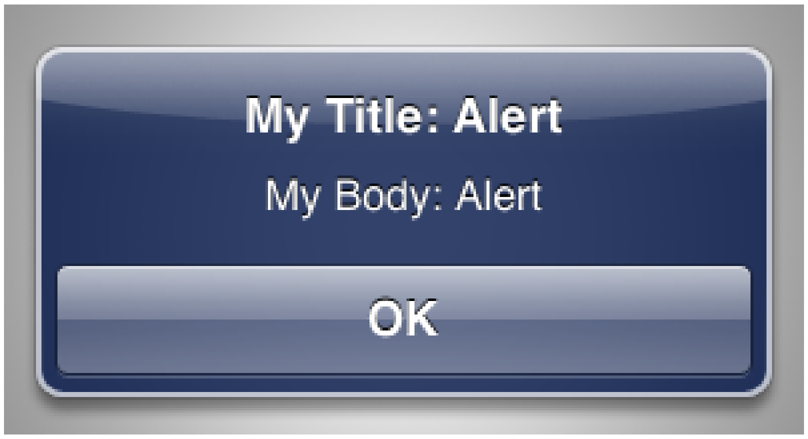
Arguments

- **number** (*string*) – number

`alert` Platforms: Android, iOS

Show a dialog window with text content that the user can dismiss. Similar to using `window.alert` except you can control the title text for the dialog.

To create an alert like this:



You would use this code:

```
forge.notification.alert("My Title: Alert", "My Body: Alert", function () {
  // ... implement logic for when user dismissed alert
});
```

`notification.`**`alert`**(*title*, *body*, *success*, *error*)

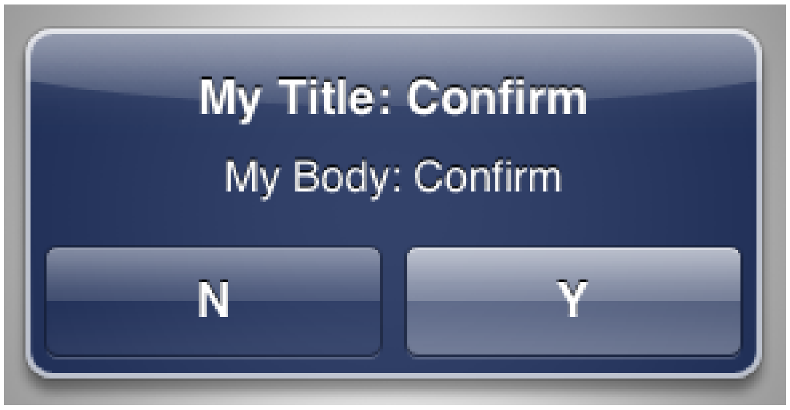
Arguments

- **title** (*string*) – text to show as the title of the dialog
- **body** (*string*) – text to show as the body of the dialog
- **success** (*function()*) – called when the user dismisses the alert
- **error** (*function(content)*) – called with details of any error which may occur

`confirm` Platforms: Android, iOS

Show a dialog window prompting the user with a "yes"/"no" style question.

To show a confirmation dialog like this:



You would use this code:

```
forge.notification.confirm("My Title: Confirm", "My Body: Confirm", "Y", "N", function (userClickedYe
  if (userClickedYes) {
    // ... implement logic for when user clicked "Y"
  } else {
    // ... implement logic for when user clicked "N"
  }
});
```

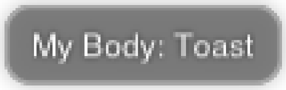
`notification.`**`confirm`**(*title*, *body*, *positive*, *negative*, *success*, *error*)

Arguments

- **title** (*string*) – text to show as the title of the dialog
- **body** (*string*) – text to show as the body of the dialog
- **positive** (*string*) – text to use for the positive action button
- **negative** (*string*) – text to use for the negative action button
- **success** (*function(result)*) – called with `true` if the user selected the positive option, `false` if they selected the negative option.
- **error** (*function(content)*) – called with details of any error which may occur

`toast` Platforms: Android, iOS

Create a small popup which disappears after a few seconds.



`notification.`**`toast`**(*body*, *success*, *error*)

Arguments

- **body** (*string*) – body
- **success** (*function()*) – called if the toast is displayed successfully
- **error** (*function(content)*) – called with details of any error which may occur

Permissions On Chrome this module will add the `notifications` permission to your app, users will be prompted to accept this when they install your app.

On Android this module will add the `VIBRATE` permission.

`parameters`: Custom config options `parameters` are custom config options your app can access via `forge.config.modules.parameters`.

Config

```
{
    "modules": {
        "parameters": {
            "my_option": "my data"
        }
    }
}
```

`prefs`: Preferences Preferences are used to save state in your extension. State is persisted between restarts and is consistent across all parts of your app.

Config The `prefs` module must be enabled in `config.json`

```
{
    "modules": {
        "prefs": true
    }
}
```

API

`get`
`prefs.`**`get`**(*name*, *success*, *error*)

Arguments

- **name** (*string*) – the name of the preference you want to query
- **success** (*function(value)*) – will be invoked as the preference value as its only parameter
- **error** (*function(content)*) – called with details of any error which may occur

Platforms: All

If the preference has not been set (with *set*), and there is no default value for this preference, `null` is returned.

`set` Platforms: All

`prefs.`**`set`**(*name*, *value*, *success*, *error*)

Arguments

- **name** (*string*) – the name of the preference you want to save
- **value** – the value to save
- **success** (*function()*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

The preference value given here will override a default value (if one was given).

`clear` Platforms: All

`prefs.`**`clear`**(*name*, *success*, *error*)

Arguments

- **name** (*string*) – the name of the preference to un-set
- **success** (*function()*) – a callback to be invoked (with no arguments) when the operation is complete
- **error** (*function(content)*) – called with details of any error which may occur

Un-sets the given preference name, so that future calls to *set* will return `undefined` (or the default preference value, if given).

`clearAll` Platforms: All

`prefs.`**`clearAll`**(*success*, *error*)

Arguments

- **success** (*function()*) – a callback to be invoked (with no arguments) when the operation is complete
- **error** (*function(content)*) – called with details of any error which may occur

Un-sets all preference names, so that calls to *set* will return `undefined` (or the default value for a preference, if given).

`keys` Platforms: All

`prefs.`**`keys`**(*success*, *error*)

Arguments

- **success** (*function(keysArray)*) – invoked with an array of the set key names as its only argument
- **error** (*function(content)*) – called with details of any error which may occur

Find which preferences have been set.

`request:` Cross-domain requests Normal in-page JavaScript is only able to make HTTP requests to the same server as one hosting the current web page. These methods allow you to work around this restriction.

Config

Note: For security reasons, you must specify the remote URLs you will send requests to in the `permissions` array of your JSON configuration file.

This array, like the `patterns` array, should contain Match Patterns to match the remote URLs you want to interact with.

To protect your users, make these match patterns as restrictive as possible.

```
{
    "modules": {
        "request": {
            "permissions": ["https://trigger.io/*"]
        }
    }
}
```

API

`get` Platforms: All

`request.get` (*url*, *callback*, *error*)

Arguments

- **url** (*string*) – the URL to GET
- **callback** (*function(content)*) – called with the retrieved content body as the only argument
- **error** (*function(content)*) – called with details of any error which may occur

The callback function *callback* is invoked with the content body of the requested URL. JSON-encoded content will automatically be parsed into a JavaScript object.

As it is limited to `GET` requests and lacks the more advanced options of *forge.request.ajax*, it's recommended that *forge.request.get* is only used in very simple scenarios.

`ajax` Platforms: All

`request.ajax` (*options*)

Arguments

- **options** (*object*) – jQuery-style parameters to control the request

Note: unlike jQuery, we expect the URL for the the request to be passed into the options hash, *not* as a positional parameter

This function is closer to the jQuery.ajax method than *forge.request.get*. However, the full range of jQuery options are **not supported** for this method, due to the structure of browser and mobile apps.

Also, note that the `error` and `success` callbacks are **not** passed a jQuery XHR object.

Currently supported options:

- accepts

- cache
- contentType
- data
- dataType
- error
- password
- success
- timeout
- type
- url
- username
- files (Mobile only, see *forge.file*)
- fileUploadMethod
- headers

Error object properties:

- `statusCode`: Status code returned from the server.
- `content`: Content returned from the server (if available).

Error values (see *error callback docs* for more detail):

- `type`: `"UNAVAILABLE"`
- `subtype`: `"NO_INTERNET_CONNECTION"` No internet connection is currently available (on iOS it is required you inform the user of this if it impacts their current experience).

Example:

```
forge.request.ajax({
  type: 'POST',
  url: 'http://my.server.com/update/',
  data: {x: 1, y: "2"},
  dataType: 'json',
  headers: {
    'X-Header-Name': 'header value',
  },
  success: function(data) {
    alert('Updated x to '+data.x);
  },
  error: function(error) {
    alert('Failed to update x: '+error.message);
  }
});
```

You can control the name of uploaded files by setting the `name` attribute, e.g.:

```
myFile.name = 'name_of_input';
forge.request.ajax({
  type: 'POST',
  url: 'http://my.server.com/upload/',
  files: [myFile],
  success: function(data) {
    alert('Uploaded file as '+myFile.name);
  },
  error: function(error) {
    alert('Failed to upload file: '+error.message);
  }
});
```

If you need to POST an image as the whole request body, use `fileUploadMethod`. E.g.:

```
forge.request.ajax({
  type: 'POST',
  url: 'http://my.server.com/upload_image/',
  fileUploadMethod: "raw",
  success: function(data) {
    alert('Uploaded image');
  }
});
```

In this example, the `Content-Type` header will be set to `image/jpeg` and the POST body will consist of just the image data with no extra encoding. This is useful in conjunction with services like Parse.

Permissions On Chrome this module will any of the Match Patterns you specify to your app, users will be prompted to accept this when they install your app.

`requirements`: App requirements The `requirements` module allows you to set specific device requirements for your apps.

Config

```
{
    "requirements": {
        "android": {
            "minimum_version": "6",
            "disable_ics_acceleration": true
        },
        "ios": {
            "minimum_version": "4.3",
            "device_family": "iphone"
        },
        "chrome": {
            "content_security_policy": "script-src 'self' https://ssl.google-analytics.com; object-s
            "web_accessible_resources": [
                "background.jpg"
            ]
        }
    }
}
```

Android

- `minimum_version`: the minimum Android API level you want to support, it must be between 5 and 15. More details can be found on the Android developers site: http://developer.android.com/guide/appendix/api-levels.html.

- `disable_ics_acceleration`: Disables hardware acceleration on Android 4.0, this is a workaround to potential rendering issues which can affect some apps on this particular version of Android.

iOS

- `minimum_version`: The iOS version is the minimum iOS version you want to support, between 4.0 and 5.1.
- `device_family`: Used to limit the types of device which can run the app, must be one of `any`, `iphone` or `ipad`.

Chrome The Chrome options relate to Chromes "manifest_version": 2 changes, which are documented on the Chrome website http://code.google.com/chrome/extensions/manifestVersion.html. The available settings are:

- `content_security_policy`: This determines what javascript can be executed in pages that belong to your extension (such as popups). The example given above is how you would allow Google Analytics to work within an extension. More documentation is available on the Chrome site http://code.google.com/chrome/extensions/contentSecurityPolicy.html
- `web_accessible_resources`: This is an array of any files in your extension which are to be accessed from external sites. The most common use of this is if your extension uses a content script to load images from your extension into a 3rd party site.

`tabs`: Tabs Management Tabs management provides functionality specific to tabs.

Config The `tabs` module must be enabled in `config.json`

```
{
    "modules": {
        "tabs": true
    }
}
```

API

`open` Platforms: All except web

Opens a new tab with the specified url with an option to retain focus on the calling tab.

On mobile this will display a *modal view* and the success callback will be called with an object containing a url and optionally a userClosed boolean property.

`tabs.`**`open`**(*url*[, *keepFocus*], *success*, *error*)

Arguments

- **url** (*string*) – The URL to open in the new tab
- **keepFocus** (*boolean*) – (optional) If true keeps the current tab focused
- **success** (*function(object)*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

`openWithOptions` Platforms: All except web

As `open` but takes an object with the following parameters:

Required:

- `url`: Required URL to open

Browser only:

- `keepFocus`: Whether or not to keep focus on the current page.

Mobile only:

- `pattern`: Pattern to close the modal view, see *modal views* for more detail.
- `title`: Title of the modal view.
- `tint`: Colour to tint the top bar of the modal view. An array of four integers in the range [0,255] that make up the RGBA color. For example, opaque red is [255, 0, 0, 255].
- `buttonText`: Text to show in the button to close the modal view.
- `buttonIcon`: Icon to show in the button to close the modal view, if `buttonIcon` is specified `buttonText` will be ignored.
- `buttonTint`: Colour to tint the button of the top bar in the modal view.

Example:

```
forge.tabs.openWithOptions({
  url: 'http://my.server.com/login/',
  pattern: 'http://my.server.com/loggedin/*',
  title: 'Login Page'
}, function (data) {
  forge.logging.log(data.url);
});
```

`tabs.`**`openWithOptions`**`(`*options*, *success*, *error*`)`

Arguments

- **options** (*object*) – Object containing url and optional properties.
- **success** (*function(object)*) – callback to be invoked when no errors occurs
- **error** (*function(content)*) – called with details of any error which may occur

`closeCurrent` Platforms: Browser only

Restrictions: Only available inside a page (not from a background script)

Close the tab which makes the call.

`tabs.`**`closeCurrent`**`(`*error*`)`

Arguments

- **error** (*function(content)*) – called with details of any error which may occur

Permissions On Chrome this module will add the `tabs` permission to your app, users will be prompted to accept this when they install your app.

`tools`: Miscellaneous tools

Config The `tools` module must be enabled in `config.json`

```
{
    "modules": {
        "tools": true
    }
}
```

API

`UUID` Platforms: All

A UUID is a globally unique token; when represented as a string, they look something like `18ADF182-7B12-4FA1-AF0B-6032108C0AE8`. Forge already uses UUIDs internally to ensure your extension doesn't conflict with others; this method returns a new UUID for you to use as a unique token.

Note: This function is synchronous and returns a value rather than taking a callback.

`tools.UUID()`

Returns a string representation of the UUID

`getURL` Platforms: All

Resolve this name to a fully-qualified local or remote resource

`tools.getURL(`*name*, *callback*, *error*`)`

Arguments

- **name** (*string*) – unqualified resource name, e.g. `my/resource.html`
- **callback** (*function(url)*) – will be invoked with the URL as its only parameter
- **error** (*function(content)*) – called with details of any error which may occur

`update_url`: Automatic update urls *Browsers only.*

URLs to check for application updates from.

Note that Firefox update URLs must begin with `https`: https://developer.mozilla.org/en/Install_Manifests#updateURL.

Config

```
{
    "modules": {
        "update_url": {
            "chrome": "url",
            "firefox": "url"
        }
    }
}
```

Release Notes This file contains information about new features and capabilities of Forge versions, along with migration information for how to upgrade from one level to another.

In your `config.json` file, you can use a major version (like `v1.3`), which means you will receive rolling updates and fixes, or you can use a minor version (like `v1.3.2`), which will only be updated with critical fixes and security patches.

To see the minor version used to create a particular build, look in `.template/platform_version.txt` in your app directory (`.template` sits alongside `src`).

v1.4

Supported Targets

- Android
- iOS
- Windows Phone
- Chrome
- Firefox
- Safari
- Internet Explorer
- Web

Changes from v1.3 The v1.4 major version changes the format of config.json slightly, putting the `orientations` configuration inside a new `display` module.

In addition, due to internal changes in how Forge apps work, you will no longer be able to make cross-domain requests without using the `forge.request` module or CORS.

Upgrade Instructions To upgrade from v1.3 to v1.4, we provide a command which automates the process of updating your `config.json` file.

If you're using the command-line tools, just run `forge migrate`: we will create a backup of your current `config.json` file in `src/config.json.bak`.

You should also check your code is not attempting to make cross-domain XHRs: either use `forge.request` instead (recommended), or CORS if you prefer.

v1.4.42 Released: 23rd April 2013

Features:

- support for wildcard Ad Hoc provisioning profiles

Bug fixes:

- saveURL and cacheURL can be safely used in a tight loop - generated file names are unique
- saveURL persists files properly on iOS (fixing regression in v1.4.41)

v1.4.41   Released: 19th April 2013

Features:

- new camera module: *modules-camera* - more reliable image capture on low memory Android devices
- improved barcode module: option to turn on light, and barcode type is available - *barcode: Barcode / QR Code scanner*

Bug fixes:

- saveURL and cacheURL in *file* module paused JS execution on iOS
- apply Reload updates asynchronously to accelerate app resumes
- focus was being lost from iOS inputs due to automatic invocation of launchimage.hide
- running apps on web target was failing with `builtin_function_or_method object has no attribute __getitem__`
- updated Android Facebook SDK to version 3.0.1 (to fix http://stackoverflow.com/questions/15877837/trigger-io-facebook-authorize-error-callback-not-called-when-user-presses-back)

v1.4.40   Released: 9th April 2013

Bug fixes:

- fixed regression of a bug in the `tabs` module on Android 2.x, lost in merge into v1.4.37

v1.4.39   Released: 8th April 2013

Features:

- audio playback API: *media: Media file playback*
- native alert, confirm and toast dialogs: *notification: Notifications*

Bug fixes:

- `web` target applications can be deployed on Node.js version 0.10
- iOS returns `purchaseState` parameter in payment callback: *payments: In-app payments* (original `PurchaseState` parameter kept for compatibility)
- non-ASCII characters are handled properly in the app name
- fixed an empty view being shown just before the launch image on Android
- data in `parameters` module available as `forge.config.modules.parameters` once again

v1.4.38   Released: 3rd April 2013

Bug fixes:

- fix for backwards compatibility: allow module configuration to set to `false`. **NB** this *does not* disable the module, however! To disable a module, remove it from `config.json` or uncheck its checkbox in the Toolkit App Config.

v1.4.37 Released: 3rd April 2013

Note: There was significant internal refactoring in this platform version: some functionality which previously worked but was not explicitly supported has been removed. In particular, `forge.ajax` does not exist; neither do modules which don't work on particular build targets (e.g. *topbar: Native top bar* on web).

Features:

- update to Parse Android SDK v1.2.3 (https://parse.com/questions/androidcontentreceivercallnotallowedexception-when-registering-for-push-notifications)
- launch images are always hidden after 5 seconds, to prevent apps appearing to hang for slow resources

Bug fixes:

- file module respects Reload updates when returning local URLs: *file: File and Camera access*
- Flurry custom events are properly sent: *flurry: Analytics with Flurry*

v1.4.36 Released: 27th March 2013

Bug fixes:

- re-installing iOS IPAs on top of existing installations caused hangs on launchimage in some situations

v1.4.35 Released: 20th March 2013

Features:

- can set minimum required iOS version to be 6.0: *requirements: App requirements*
- allow the *web* target Node.js app to be deployed at non-root paths (http://stackoverflow.com/questions/15070765/)

Bug fixes:

- fix crash in API demo (https://github.com/trigger-corp/Forge-API-Demo/issues/7)
- fix playback of video and audio after Reload usage on iOS - note the Gotchas on *Using Trigger.io Reload*!

v1.4.34 Released: 12th March 2013

Bug fixes:

- fix slight inaccuracy when adding calendar events: *modules-calendar*
- fix race condition where tabbar buttons could be wrongly ordered: *tabbar: Native tab bar*
- fix modal view redirect on older versions of Android (http://stackoverflow.com/questions/15262840/)
- fix NullPointerException in urlhandler module *urlhandler: Custom URL schemes* (http://stackoverflow.com/questions/13824961/)
- fix cross-device link error (http://stackoverflow.com/questions/11578443/)

v1.4.33 Released: 6th March 2013

Bug fixes:

- application of Reload updates was broken on iOS devices

v1.4.32 Released: 4th March 2013

Features:

- support for subscription payments on Android: see *payments: In-app payments*
- support for using the `forge` APIs on trusted remote HTML pages: see *Configuration for your app*

Bug fixes:

- update to Android Parse SDK version 1.1.15 to fix http://stackoverflow.com/questions/14811733/

v1.4.31 Released: 28th February 2013

Features:

- new `installationInfo` API for Parse, by popular demand: see *push: notifications (with Parse)*
- pause Reload updates, and receive progress updates: see *reload: Push updates to deployed apps*
- create file fixtures when developing native plugins: see *Working with files*
- update Parse SDK to version 1.1.32

Bug fixes:

- benign stack trace on startup from `urlhandler` module
- enhanced date inputs on Android now fire on touchend rather than touchstart (http://stackoverflow.com/questions/14551349/)
- fix `minimum_version` requirement for iOS

v1.4.30 Released: 15th February 2013

Bug fixes:

- handle Android gallery not including Exif data in photos

v1.4.29 Released: 30th January 2013

Bug fixes:

- sensible fallback if image processing fails on Android

v1.4.28 Released: 30th January 2013

Bug fixes:

- include `ForgeFile.h` in native plugin Inspector projects on iOS

v1.4.27 Released: 29th January 2013

Features:

- Exif orientation data is used when displaying or uploading images on Android
- launch IE as original user after extension installation
- prefix plugin projects with name in Eclipse
- update Parse Android SDK to version 1.1.11

Bug fixes:

- `forge.request` was interacting badly with Reload in some situations on Android
- fix threading issues in *barcode* and Catalyst
- Parse broadcast channel was broken on Android

v1.4.26 Released: 17th January 2013

Note: Due to the switch to using Gson, the way to return non-primitive results from native plugins has changed: see *Communicating with native code*

Features:

- create calendar events with the *calendar module*
- Use Gson for JSON parsing and serialisation for increased performance on Android
- new `forge.file.saveURL` API: *file: File and Camera access*

Bug fixes:

- IE activates properly on pages opened with `target="_blank"`
- Android datepicker activates for `datetime-local` inputs
- Android datepicker results are properly zero-padded

v1.4.25 Released: 14th January 2013

Bug fixes:

- support for large Android launchimages
- fix for NumberFormatException in Android contacts module
- `facebook.ui` result now has same schema as the JavaScript SDK

v1.4.24 Released: 18th December 2012

Features:

- you can run the iOS simulator at a specified version with `simulatorfamily` and `simulatorsdk` - see *Configuration for the tools*

Bug fixes:

- Android launchimages are scaled properly on high pixel density screens
- HTTP 401 does not cause NullPointerException on Android when no username and password supplied

v1.4.23 Released: 7th December 2012

Features:

- server-side code signing for IE extensions
- Android native date picker fires `blur` event when complete
- during development on Windows or Linux, iOS apps are only partially code-signed for performance

Bug fixes:

- fullscreen display didn't work for holo theme Android devices

- Android native date picker follows W3C spec when returning values
- `facebook.ui` returns dialog outcome information on iOS

v1.4.22 Released: 30th November 2012

Features:

- support for IE 10 extensions

Bug fixes:

- Android native date picker results were off-by-one on the month
- unicode characters in app description caused build failures on some platforms
- running the "web" target repeatedly would cause address in use errors on OS X

v1.4.21 Released: 21st November 2012

Features:

- ability to set the background color behind Android launch images (*docs*)

Bug fixes:

- incorrect data was returned for emails by the contacts API on Android
- handle usage of unavailable APIs more gracefully

v1.4.20 Released: 7th November 2012

Features:

- cookies are persisted by default on Android
- Windows Phone builds are now done against the version 8 SDK
- launch image can be hidden manually (*docs*)
- support for iOS 6.1 beta
- native date / time pickers for Android (*docs*)

Bug fixes:

- fix issue where only Google contacts were returned by `contact.selectAll`
- modal views wouldn't close when user hit back button on Android

v1.4.19 Released: 29th October 2012

Features:

- Command-line tools bundled in Toolkit can update the Toolkit install
- native plugins v1 - see *Native plugins*
- Flurry analytics module: see *docs*
- update to Firefox Addon SDK 1.10
- ability to manually quit the app when the back button is pressed on Android - see *event: Events*

v1.4.18 Released: 15th October 2012

Bug fixes:

- “publish” permissions work properly with new Facebook SDK on iOS 6

v1.4.17 Released: 12nd October 2012

Features:

- support for using Linux for iOS builds: *Developing iOS apps on Linux*
- true native back buttons for the topbar module on iOS: *topbar: Native top bar*
- update to version 3.1.1 of the Facebook SDK for iOS for *facebook: Facebook SDK access*
- new `selectAll` and `selectById` methods in *contact: Accessing contacts*
- new Facebook API to check authentication status
- support for coloured status bar on iOS 6 (`setTint` in *topbar: Native top bar*)
- ability to create and use wireless distribution manifests for iOS *Wireless distribution on iOS*

Bug fixes:

- video uploads to Facebook API were failing

v1.4.16 Released: 1st October 2012

Warning: Due to a bug to do with resource caching in iOS 6, we’ve been forced to remove the `applyNow` method from the Reload module.

Features:

- more intelligent diff made during Reload update: faster and less bandwidth consumed
- ability to build for iPad or iPhone/iPod only: *requirements: App requirements*
- post-build hooks: *Using hooks*
- hooks are passed the currently-building target as first command-line argument
- build and run iOS apps from Linux *Developing iOS apps on Linux*

Bug fixes:

- fix json2.js operation on IE9 running in IE7 compatability mode
- ability to set the same cookie several times in one request on web target
- localStorage and webSql databases are persisted correctly

v1.4.15 Released: 25th September 2012

Features:

- register custom URL schemes: *urlhandler: Custom URL schemes*
- beta of custom native plugins complete *Native plugins*

Bug fixes:

- non-ASCII characters in some config fields were causing build problems

- can run Firefox extensions automatically on Linux
- Android landscape launchimages properly used
- `null` values in multipart/form-data requests are not sent to server

v1.4.14 Released: 17th September 2012

Warning: To accommodate the iPhone 5, this platform version requires you to set the new `iphone-retina4` configuration directive in the *launchimage module*.

Features:

- support for iOS 6 and iPhone 5

Bug fixes:

- fixed canvas `drawImage` crashing when using external resources

v1.4.13 Released: 4th September 2012

Features:

- consider build timestamps while Reloading so new installs don't apply older updates
- add `node_path` local configuration option if Node.js is not on your path: *Best practices*
- programmatically control allowed app orientation: *display: App display options*

Bug fixes:

- fix POST encoding of objects in arrays http://stackoverflow.com/questions/12194600/forge-request-ajax-post-data-as-json
- fix iPad landscape-mode launchimage distortion
- IE installer uses configured icon as branding

v1.4.12 Released: 24th August 2012

Features:

- option to *disable hardware acceleration* on Ice Cream Sandwich due to some rendering issues in libraries such as KendoUI
- iOS: automatically use distribution developer certificate with distribution provisioning profile and vice versa

Bug fixes:

- updated iOS app install utility for better Mountain Lion support, faster operation and increased reliability
- Forge-based IE extensions can be disabled in IE 9
- initial connectionStateChanged event fired even earlier
- tabbar and topbar buttons aren't duplicated by Reload
- content is zoomable and pannable in Android modal views
- cookies containing double quotes work when using web target with Opera

v1.4.11 Released: 22nd August 2012

Bug fixes:

- fix Facebook API regression, where authentication flows didn't return to the app
- fix Express's zlib dependency on Heroku http://stackoverflow.com/questions/11995324/zlib-module-not-playing-nicely-with-web-deployment

v1.4.10 Released: 20th August 2012

Features:

- can set name of files uploaded through request.ajax
- better Reload download logic to speed up update deployment

Bug fixes:

- fullscreen mode incompatible with orientation limitation on iOS
- unicode characters in app config could cause problems in some situations
- prerendered icons for iOS were broken

v1.4.9 Released: 13th August 2012

Features:

- re-use of Reload files already present on iOS device

Bug fixes:

- version number updated properly in IE setup scripts
- resource loading on iOS improved using Reload
- tools.getURL needed adjustment for Reload

v1.4.8 Released: 8th August 2012

Bug fixes:

- relative resource paths in CSS files on iOS
- make AVD creation more resilient to failure
- handle lack of JRE more gracefully
- force IE popups to the foreground

v1.4.7 Released: 7th August 2012

Bug fixes:

- playback of locally bundled media files fixed on iOS
- loading locally bundled resources in modal views fixed on iOS
- fixed incompatibility between iOS contact module and MS Exchange

v1.4.6 Released: 2nd August 2012

Features:

- Facebook authentication details returned as parameter to facebook.authorize

Bug fixes:

- `minimum_version` configuration on Android was causing build problems for some
- remove dependency on Express 2.5.0 for web target
- remove default orientation configuration and fix Android "any" mode

v1.4.5 Released: 1st August 2012

Bug fixes:

- ensure focus events work properly for popup windows on IE

v1.4.4 Released: 31st July 2012

Bug fixes:

- fix internal generateQueryString method on IE

v1.4.3 Released: 26th July 2012

Bug fixes:

- creating modal dialogs was broken on some older versions of Android

v1.4.2 Released: 24th July 2012

Bug fixes:

- enable use of modal views immediately after app launch on iOS
- modules are fully disabled by default, unless explicitly enabled

v1.4.1 Released: 20th July 2012

Features:

- support retina scaled images for iPad
- integration with native Facebook SDKs
- use `enableHighAccuracy` in iOS geolocation API

Bug fixes:

- topbar and tabbar buttons are correctly re-added after app is closed on Android
- network activity indicator properly cleared after closing iOS modal views

v1.4.0 Released: 17th July 2012

Features:

- *Reload*
- lifecycle events (appPaused and appResumed *docs*)
- barcode scanning module: *barcode: Barcode / QR Code scanner*
- use Chrome manifest version 2 (see *requirements: App requirements*)
- fullscreen support (*display: App display options*)

v1.3

Supported Targets

- Android
- iOS
- Windows Phone
- Chrome
- Firefox
- Safari
- Internet Explorer
- Web

Changes from v1.2 The v1.3 platform release changes the format of config.json to put most optional configuration into separate modules, this allows Forge to provide more features without having them all enabled for every app.

By default all of the features from v1.2 will be enabled, but these can be disabled if not required. Disabled modules allow the Forge generation process to remove code from your app, making it smaller. Modules also define the permissions your app will required, so disabled unused modules will reduce the permissions users are prompted for when installing your app.

Upgrade Instructions To upgrade from v1.2 to v1.3 your `config.json` file needs to be updated, this can be done automatically by running `forge migrate` with the command line tools, or choosing to migrate from Trigger Toolkit.

The migration process will automatically update your `config.json` file to v1.3, if for any reason it doesn't work a backup of your `config.json` file will be saved as `config.json.bak`.

v1.3.23 Released: 20 July 2012

Features:

- migration script to upgrade to v1.4

v1.3.22 Released: 18th July 2012

Bug fixes:

- launchimage on iPad is correctly sized

v1.3.21 Released: 12th July 2012

Features:

- network activity spinner / progress bar shown while loading modal views

Bug fixes:

- connectionStateChanged callbacks are fired at least once
- request.ajax response contains the body data for non-200 status codes on Android

v1.3.20 Released: 12th July 2012

Bug fixes:

- re-enable running Firefox automatically
- clean up some extra files produced by new Android SDK

v1.3.19 Released: 3rd July 2012

Bug fixes:

- forge.prefs fix for Internet Explorer

v1.3.18 Released: 3rd July 2012

Features:

- allow ad-hoc builds to be created on iOS

Bug fixes:

- update to latest Parse Android SDK for push notifications fixes
- panel sizing fix for Firefox

v1.3.17 Released: 22nd June 2012

Bug fixes:

- a Python fix which makes us less incompatible with 2.6 - note 2.7 is still the only officially supported Python version!
- Windows Phone IE does not support setZeroTimeout

v1.3.16 Released: 18th June 2012

Bug fixes:

- “no such file or directory” during Android tasks on some Linux setups
- Node.js directory locking issue on Windows
- lots of Trigger Toolkit UI tweaks and fixes
- allow for running Forge builds on non-root mount point

v1.3.15 Released: 11th June 2012

Features:

- better Q & A system for Trigger Toolkit
- build for iOS on Windows: http://trigger.io/cross-platform-application-development-blog/2012/05/31/work-on-what-you-want-week-at-trigger-io/
- iframes are allowed on iOS now - embed media players, buttons and so on

Bug fixes:

- `about:blank` caused app to crash in iOS simulator
- logcat process were left hanging after runs

v1.3.14 Released: 30th May 2012

Features:

- can install apps to SD card on Android

Bug fixes:

- default value for file character encoding guess
- handle non-ASCII command line parameters
- playVideo callback is fired after video finishes and focus returns
- mailto: links handled properly in modal views

v1.3.13 Released: 22nd May 2012

Features:

- show / hide topbar and tabbar programmatically
- specify minimum version of iOS and Android
- complete `forge.file` support on Windows Phone 7
- in-app purchase support
- updated Firefox SDK

v1.3.12 Released: 17th May 2012

Features:

- `.template/platform_version.txt` created as part of build process
- button popups on IE are moved and resized intelligently

Bug fixes:

- index not required for tabbar.addButton
- large number of tabbar buttons handled properly
- callbacks firefox after tabbar and topbar buttons added

v1.3.11 Released: 15th May 2012

Features:

- disable icon glossiness on iOS (*docs*)
- `file.getLocal` and `file.string` support in non-mobile targets (*docs*)
- Catalyst shows waiting message until debugger has connected

Bug fixes:

- run app on Android emulator, when emulator has been started automatically
- prebuild hooks are found and run correctly

v1.3.10 Released: 10th May 2012

Features:

- full video support on Android and iOS
- topbar module on Windows Phone

Bug fixes:

- callbacks sometimes not invoked after tabbar.addButton
- window.forge initialisation sometimes got stuck in a loop
- NullPointerException sometimes occurring when using console.log on Android
- prevent BroadcastReceiver intent leak on Android
- prevent console windows popping up during Toolkit builds

v1.3.9 Released: 8th May 2012

Features:

- greatly improved error messages and status codes for failed HTTP requests on Android

v1.3.8 Released: 7th May 2012

Bug fixes:

- handle change in status codes returned by Heroku API

v1.3.7 Released: 6th May 2012

Features:

- Windows Phone 7 support: partial

Bug fixes:

- ensure iOS permission dialog shown on main thread: was sometimes not visible
- fix segfault which occurred in some situations showing camera on iPhone running v5.1

v1.3.6 Released: 3rd May 2012

Bug fixes:

- character encoding guessing now deals with empty files
- ensure connection change event is fired soon after app startup
- callbacks are properly fired for camera usage (iOS) and modal views (Android)
- launch images on Android

v1.3.5 Released: 2nd May 2012

Features:

- connection status information in *forge.is.connection*, as well as *connection state change events*
- Web SQL support

Warning: Web SQL is not supported in all browsers or on all devices: http://caniuse.com/#search=websql

v1.3.4 Released: 29th April 2012

Bug fixes:

- Parse push notifications were not recieved on Android in some situations

v1.3.3 Released: 27th April 2012

Features:

- styling for *modal views on mobile*
- better incremental builds: faster development cycle in normal conditions

Bug fixes:

- authentication loop occurring in some situations when deploying code to Heroku
- users cancelling out of iPad gallery now fires the error callback
- support for nested JavaScript objects sent through forge.request.ajax
- incorrect keystore password produces clearer error message

v1.3.2 Released: 19th April 2012

Bug fixes:

- handle *the native top bar* not being styleable on older iPhones
- disable troublesome Windows Phone builds temporarily

v1.3.1 Released: 17th April 2012

Features:

- *pre-build hooks*
- re-use server-side builds, improving `forge build` performance

Bug fixes:

- correct usage of `homepage`, `update_url`, `author` and `icons` entries from your config.json in various browser extension manifests
- quitting Android 2.1 app with the back button was causing app crash
- push notifications with Parse on iOS were not enabled properly
- process suspended while looking for Android device on Linux
- better handling of location permission denied after image capture on iOS

v1.3.0 Released: 5th April 2012

Features:

- *button module* on IE
- `getLocal` function in *file module*
- native bar at bottom of app: *tabbar module*
- ask for the minimum set of required permissions on Android

v1.2

Supported Targets

- Chrome
- Android
- Firefox
- iOS
- Web

v1.2.4 Released: 27th April 2012

2.3.3 Tools

This section contains reference documentation on how to use our tools to develop applications for the Forge platform.

The Trigger.io Toolkit

The Trigger.io Toolkit is a graphical interface to start / stop builds, package your app, push *Reload* updates to your users and create *native plugins*.

It offers more functionality than the *command-line tools* and is the preferred user interface for new and existing Trigger.io users.

Installing the Trigger.io Toolkit

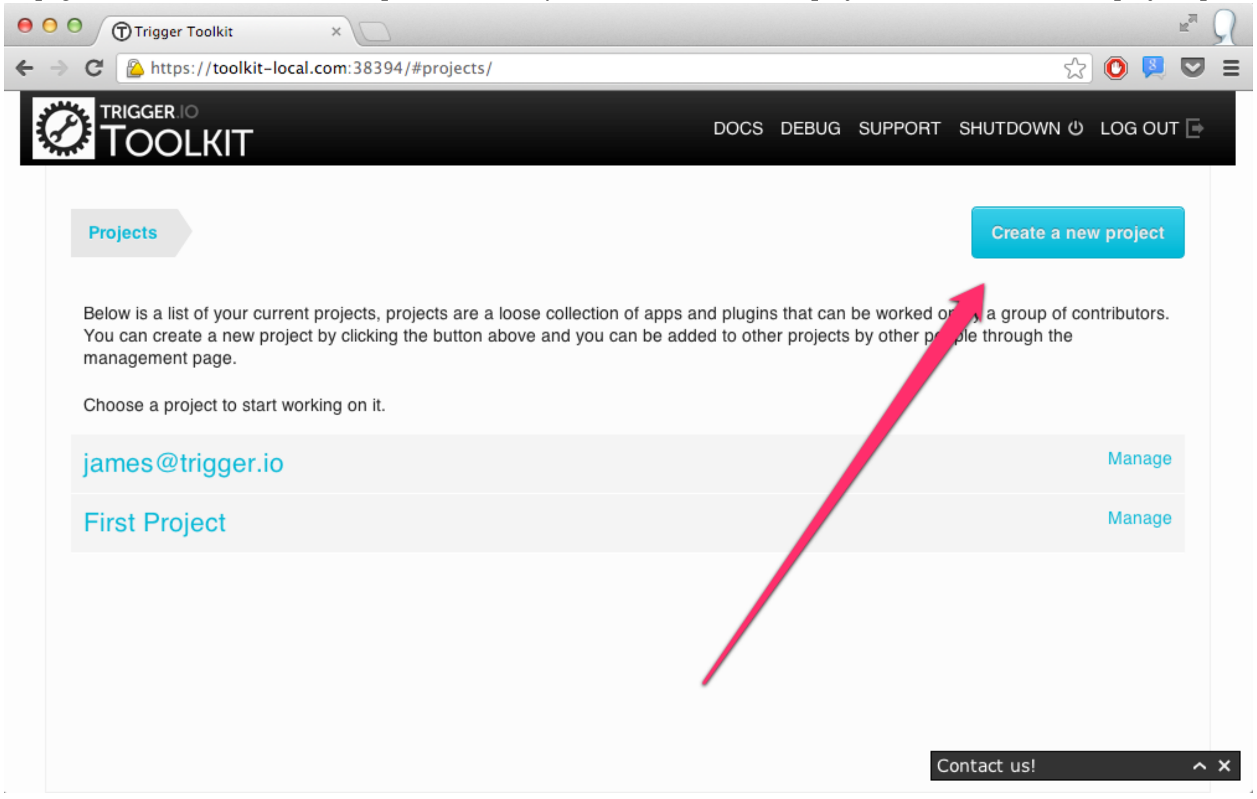
You can download the Toolkit installer from https://trigger.io/forge/toolkit/.

Follow the OS-specific instructions on the download page to install and start the Toolkit.

Common tasks in the Trigger.io Toolkit

Creating a new project Open the toolkit and click the “Create a new project” button.

A page on our website will be opened so that you can enter the new project name and select a project plan.

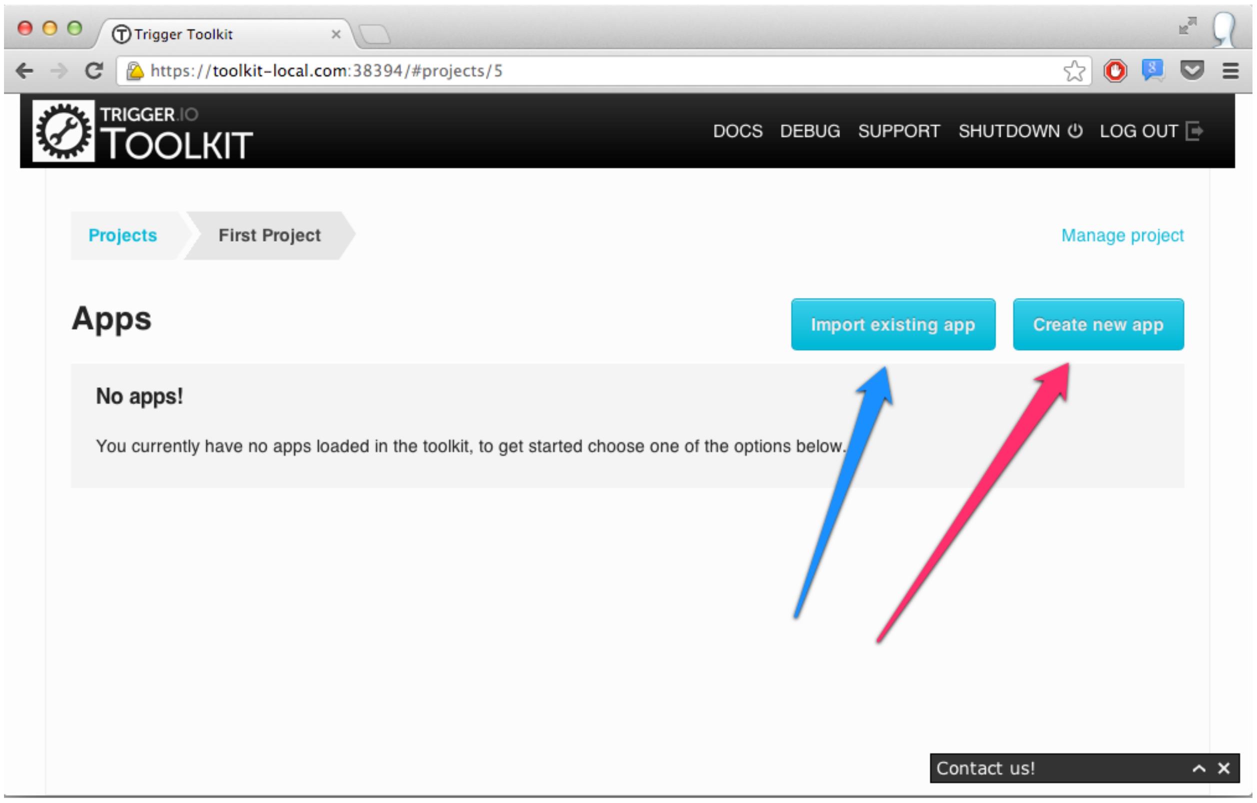


Below is a list of your current projects, projects are a loose collection of apps and plugins that can be worked on by a group of contributors. You can create a new project by clicking the button above and you can be added to other projects by other people through the management page.

Choose a project to start working on it.

Creating a new app Enter into a project by clicking on its name, then click the “Create new app” button (shown in pink).

If one of your colleagues has sent you the source code for an existing app, chose “Import existing app” (shown in blue) and navigate to where the source is on your computer.

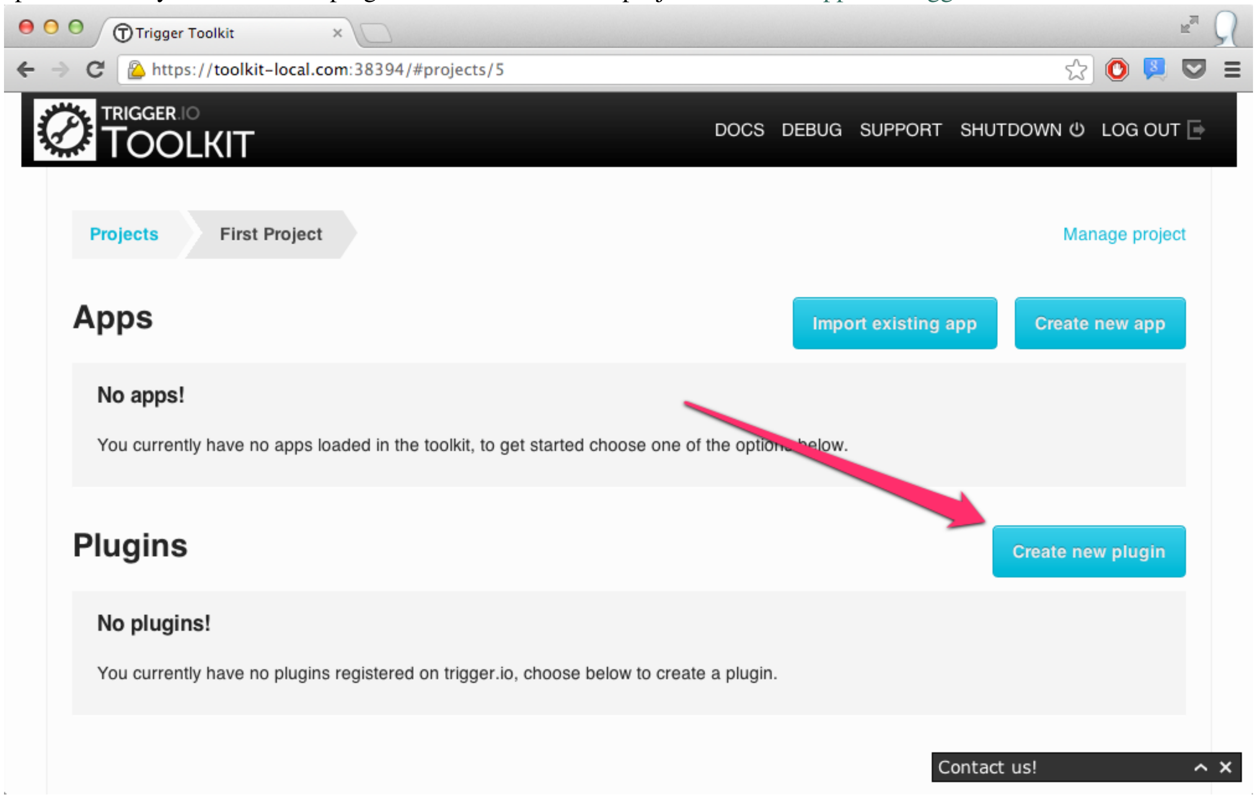


No apps!

You currently have no apps loaded in the toolkit, to get started choose one of the options below.

Creating a new plugin Inside a project, click the "Create new plugin" button.

This option will only be available if plugins are enabled for this project - contact support@trigger.io for more information.



No apps!

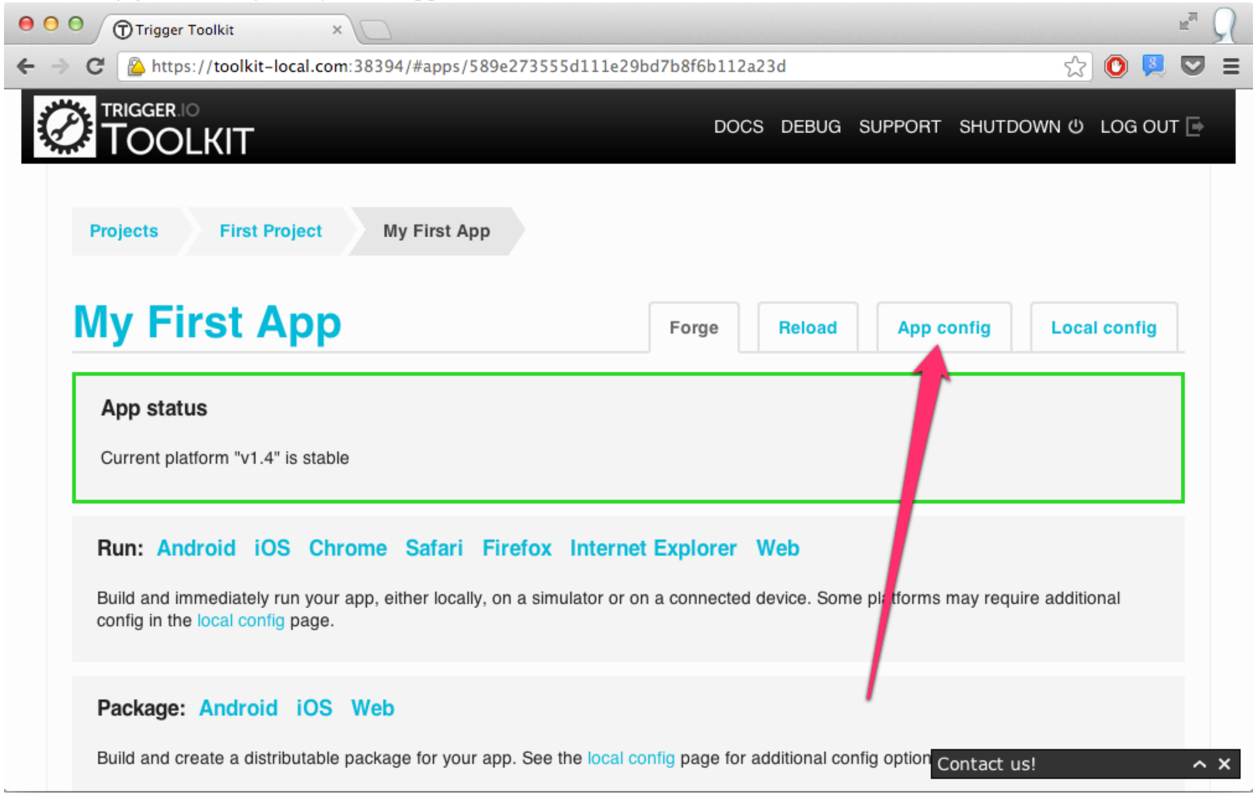
You currently have no apps loaded in the toolkit, to get started choose one of the options below.

No plugins!

You currently have no plugins registered on trigger.io, choose below to create a plugin.

Changing the configuration for an app Enter into an app by clicking on its name, then click on the "App config" tab.

See *Configuration for your app* for more information about how to use this interface.



My First App

App status

Current platform "v1.4" is stable

Run: Android iOS Chrome Safari Firefox Internet Explorer Web

Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

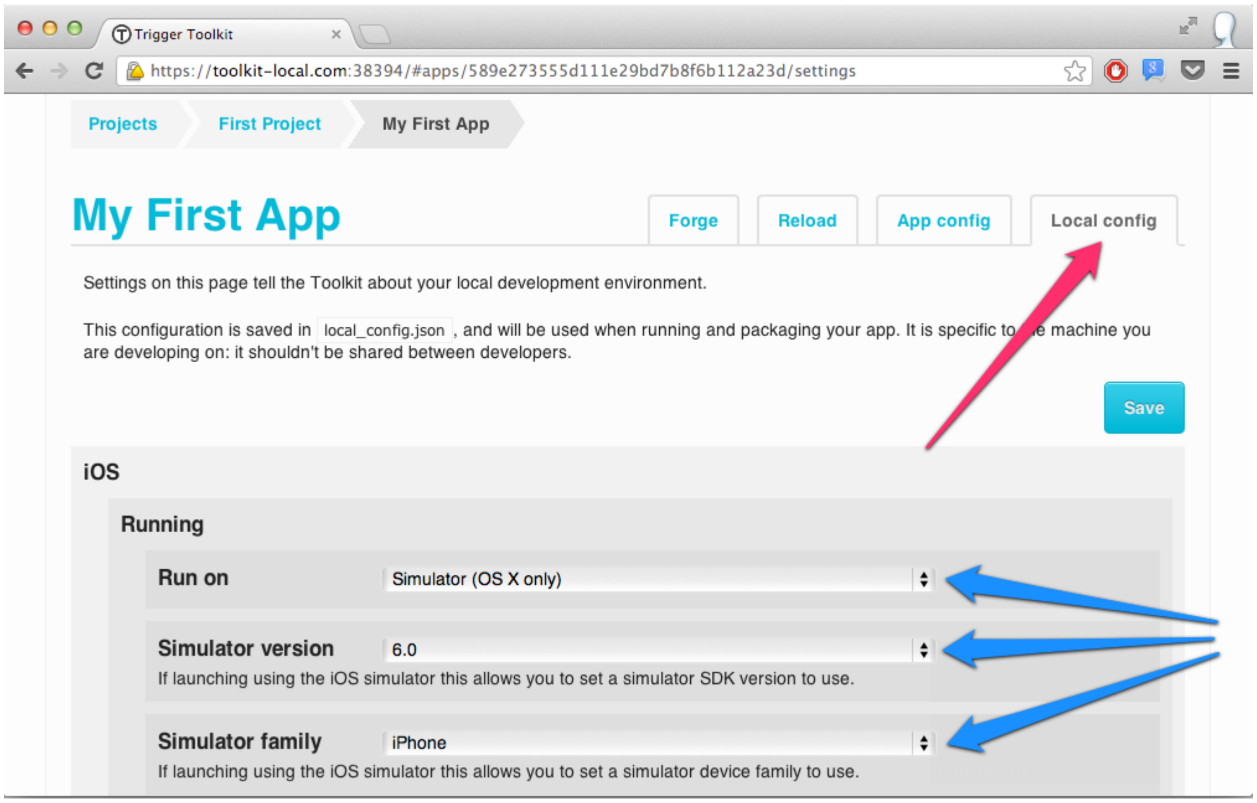
Package: Android iOS Web

Build and create a distributable package for your app. See the local config page for additional config option

Running an app on an iOS device You may want to first update your Toolkit configuration to run the app on the right device or emulator.

Enter the Local config tab (marked in pink) then update the three settings concerning running iOS apps (marked in blue).

See *Configuration for the tools* for more information about how to use this interface.



My First App

Settings on this page tell the Toolkit about your local development environment.

This configuration is saved in `local_config.json`, and will be used when running and packaging your app. It is specific to the machine you are developing on: it shouldn't be shared between developers.

iOS

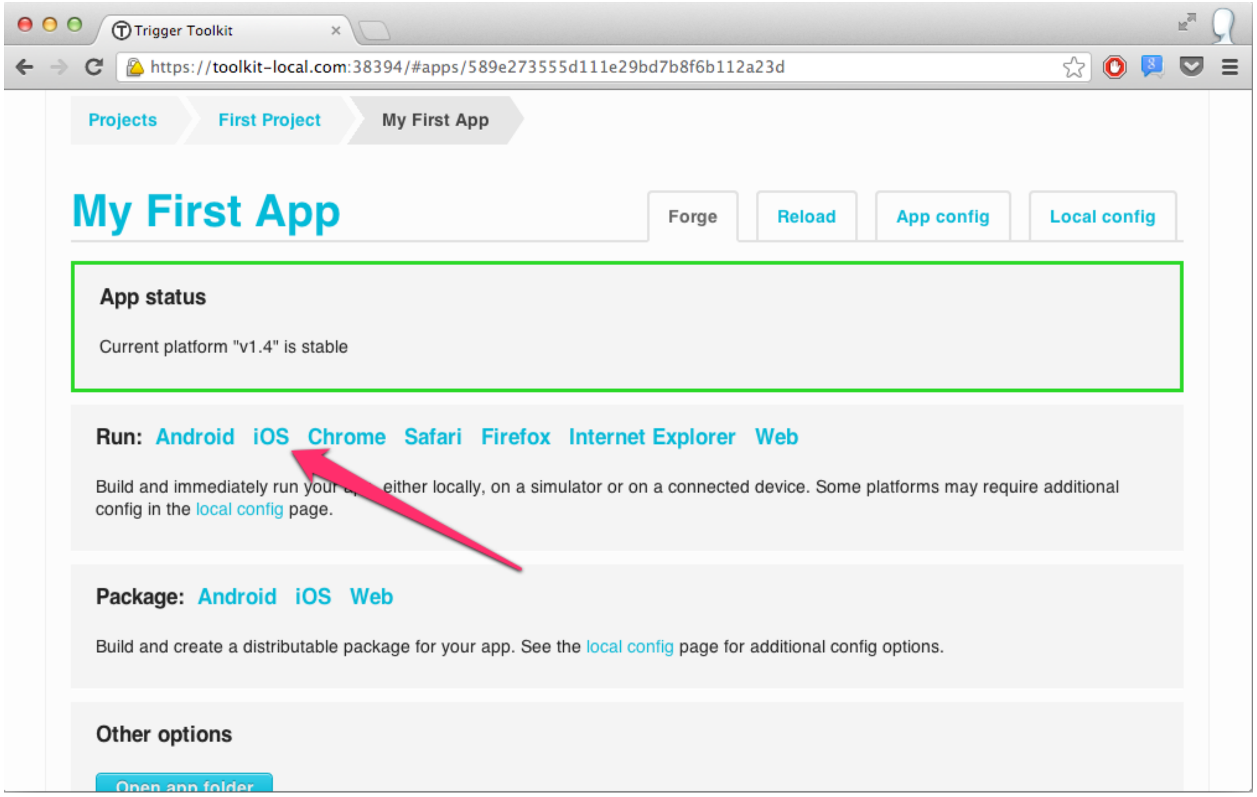
Running

Run on: Simulator (OS X only)

Simulator version: 6.0
If launching using the iOS simulator this allows you to set a simulator SDK version to use.

Simulator family: iPhone
If launching using the iOS simulator this allows you to set a simulator device family to use.

Then, click on “iOS” in the “Run” section to start the app running.



My First App

App status

Current platform "v1.4" is stable

Run: Android iOS Chrome Safari Firefox Internet Explorer Web

Build and immediately run your app either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

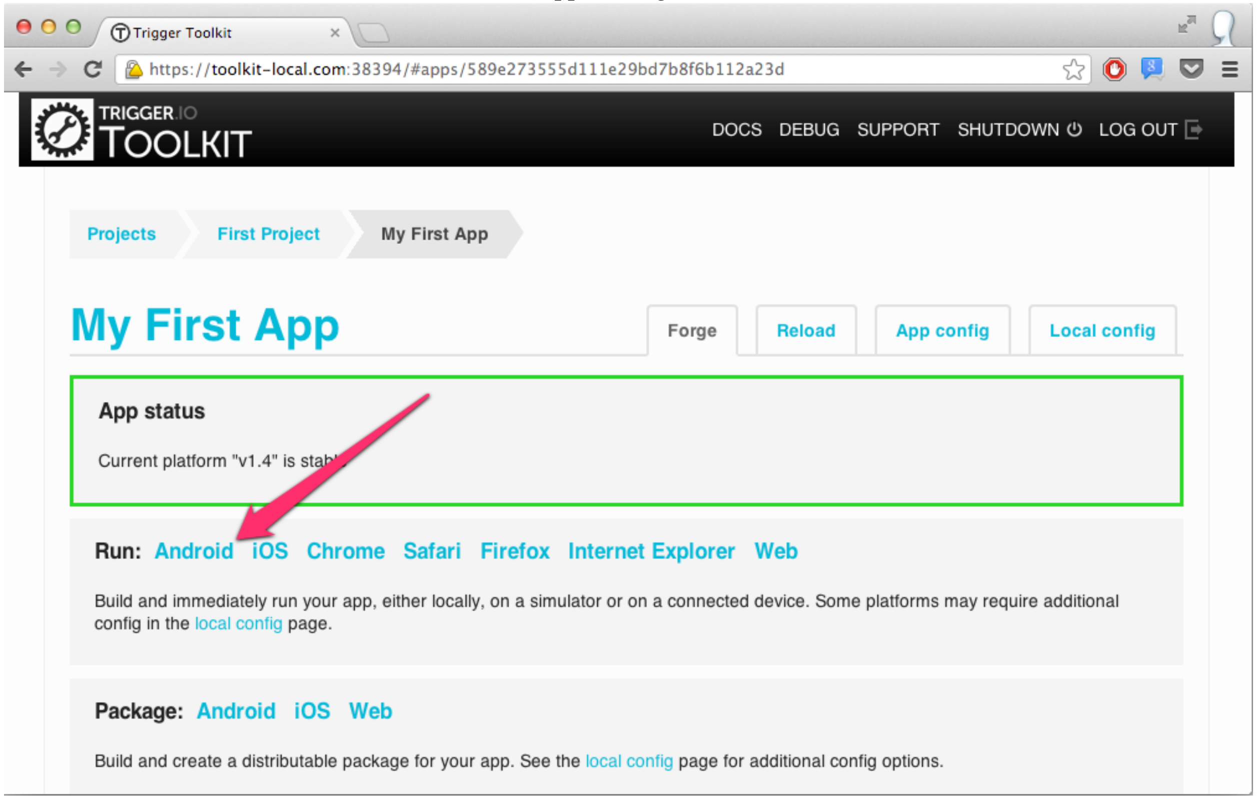
Package: Android iOS Web

Build and create a distributable package for your app. See the local config page for additional config options.

Other options

Running an app on Android In the same way as iOS, extra configuration parameters are available for Android in the Local config tab, but you normally don't need to update these.

Just click on "Android" in the "Run" section to start the app running on a device (if one is attached) or to start the emula-



My First App

App status

Current platform "v1.4" is stab

Run: Android iOS Chrome Safari Firefox Internet Explorer Web

Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

Package: Android iOS Web

Build and create a distributable package for your app. See the local config page for additional config options.

tor.

Command-line tools

The command-line tools are supplied to enable scripted usage of the Trigger.io Forge build service. For normal usage, it is expected that the *Trigger.io Toolkit* will be easier and more powerful.

There are four main commands when using Trigger.io:

- `forge create`
- `forge build`
- `forge run`
- `forge package`

Reload has a number of commands which can be run from the command line:

- `forge reload list`
- `forge reload create`
- `forge reload push`

These are explained in more detail in *Using Trigger.io Reload*.

There are also some additional commands which may be used:

- `forge check`: Perform some sanity checks on the app, including parsing the Javascript to check for syntax errors.

- `forge migrate`: Migrate to the next version of Forge, you will be prompted to run this when it is available.

Run any command with the `--help` argument to see more information about the particular command.

Note: All commands can be run with the `--verbose` parameter, to enable the display of more output.

Location of `forge` executable

To run forge commands use the forge executable in your Toolkit installation:

Windows

```
C:\> "C:\Users\<Your Username>\AppData\Local\Trigger Toolkit\forge.exe"
```

Mac users

```
$ $HOME/Library/Trigger\ Toolkit/forge
```

Linux users

```
$ ~/TriggerToolkit/forge
```

Command-line parameters

Parameters to the `forge` commands can be given as command-line options, or *stored in a file*.

Command-line options are dot-separated names, like `--android.sdk /path/to/android-sdk`.

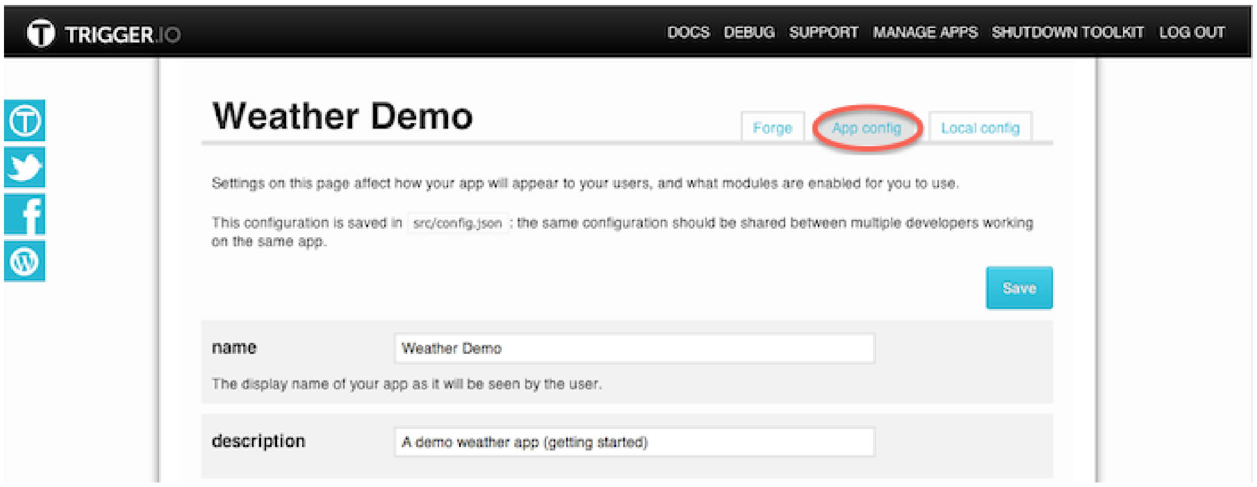
A complete list of command-line options, is given in *Available Forge Parameters*.

Configuration for your app

The configuration of your app is stored in a file called `config.json`. It determines basic settings for your app, such as the name users will see, as well as module config which allows various parts of Forge to be enabled or configured individually.

Modifying `config.json`

You can edit your app config through the Toolkit UI by clicking the App config tab in the top right of the app screen:



Weather Demo

Settings on this page affect how your app will appear to your users, and what modules are enabled for you to use.

This configuration is saved in `src/config.json` : the same configuration should be shared between multiple developers working on the same app.

name	Weather Demo
description	A demo weather app (getting started)

The display name of your app as it will be seen by the user.

Alternatively, you can edit `config.json` directly using your preferred text editor: it is located in the `src` directory. Be careful to maintain correct JSON syntax whilst doing this.

Modules

Most of the configuration for apps is available in the form of modules, you can find a list of modules *in these docs*. Each module your app uses needs to be included in your `config.json`, some modules can simply be given the value `true` to be enabled, others require their own configuration options. Each modules configuration options can be found within its documentation page.

Note: To disable a module, uncheck its checkbox in the Toolkit App Config view, or remove it from your `config.json` file: setting a module's config value to `false` **doesn't** disable the module.

Field summary

Below is a template of a basic `config.json` file with links to a detailed description of each field.

```
{
    "name": "My App",
    "author": "Forger",
    "description": "My First Forge App.",
    "version": "1.0",
    "platform_version": "v1.3",
    "homepage": "http://example.com"",
    "modules": {},
    "partners": {},
    "config_version": "2",
    "trusted_urls": [ "http://example.com/use_forge/*" ]
}
```

Fields

This section includes more detailed information on the contents of each field, with links to other documentation where appropriate.

name This will be the name for your app, a short, descriptive name is recommended as in some situations long names may be cut off.

author This text will be displayed as the author or creator of the app, depending on the platform.

description *Optional*.

A longer description of what your app does. This description may be displayed to users during and after installation, to let them know what the app does.

version The version of your app. It must be formatted as up to three dot-separated numbers, e.g. `1.1` or `0.99.9`.

platform_version As the Forge platform grows and improves, we may deprecate and remove some functionality. To prevent these updates from breaking your app, use this field to specify the version of the Forge platform you wish to build on top of. See *Release Notes* for more information about platform versions.

homepage *Optional*.

Your website, or location of more information about this app.

partners Configuration for 3rd party integration. For more information check *our partners*.

modules Enable and optionally configure optional modules. For more information check *individual modules*.

config_version An internally used reference to keep track of changes to the Forge config file schema, you shouldn't need to change this property manually.

trusted_urls **Mobile only**

An array of trusted external URL match patterns. If your navigates to a URL matching one of these patterns, JavaScript on that page will be able to use the `forge` APIs.

Configuration for the tools

Configuration settings concerning things like developer certificates shouldn't be shared across a whole team. This local configuration is stored in a file called `local_config.json`, and can be updated through the "Local Config" tab in the Trigger Toolkit.

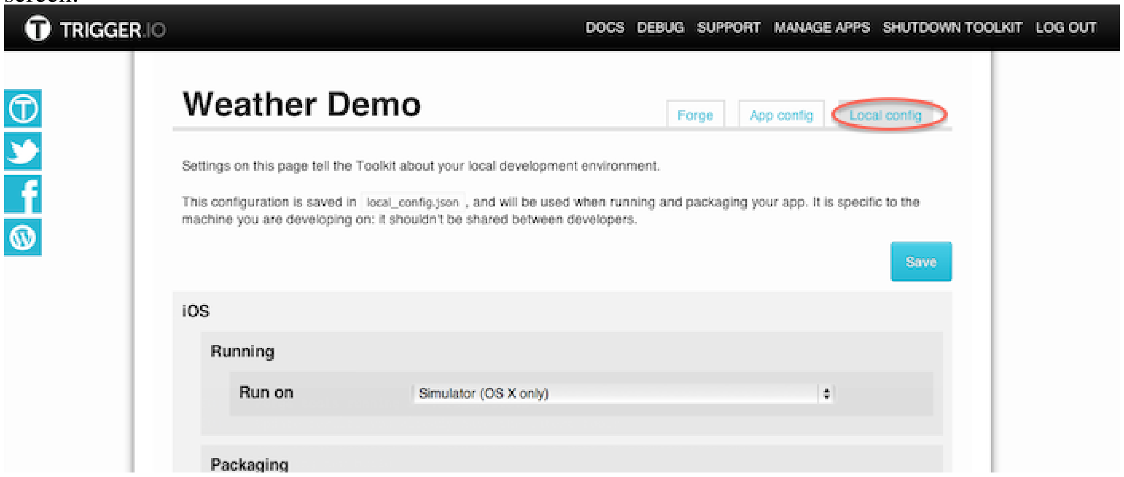
The file must be located along side the `src/` directory, for example:

```
my-app/
    development/
    src/
    local_config.json
```

If you're using the command-line, all local config can be overriden with command-line switches too.

Modifying `local_config.json`

You can edit your local configuration through the Toolkit UI by clicking the Local config tab in the top right of the app screen:



Weather Demo

Settings on this page tell the Toolkit about your local development environment.

This configuration is saved in local_config.json , and will be used when running and packaging your app. It is specific to the machine you are developing on: it shouldn't be shared between developers.

Run on: Simulator (OS X only)

Alternatively, edit the file directly using your preferred text editor.

Format of `local_config.json`

`local_config.json` is a JSON file which specifies various runtime parameters of the `forge` commands.

There is a *general* section, for parameters which are not linked to any particular target, and then per-target sections: *ios*, *android* and so on:

```
{
  "general": {
              "reload": {
                      "external": true
              }
  },
  "ios": {
    "device": "simulator",
    "profiles": {
      "DEFAULT": {
        "provisioning_profile": "/home/trigger/development.mobileprovision"
      },
      "release": {
        "provisioning_profile": "/home/trigger/release.mobileprovision",
        "developer_certificate": "iPhone Distribution",
        "developer_certificate_path": "C:\developer.pfx",
        "developer_certificate_password": "myp4ssw0rd"
      }
    }
  },
  "android": {
    "sdk": "/opt/local/android-sdk",
    "profiles": {
      "DEFAULT": {
        "keystore": "/home/trigger/development.keystore",
        "keyalias": "trigger"
      },
      "release": {
        "keystore": "/home/trigger/release.keystore",
        "keyalias": "trigger"
      }
    }
  }
}
```

A note on file paths You can specify files using a path relative to the current directory, or you can use an absolute path.

On Windows, you will need to escape the backslashes in paths for your configuration file, e.g.:

```
{
    "andoid": {
        "sdk": "C:\\Users\\Tim Monks\\android-sdk-windows"
    },
    "profiles": {
        "DEFAULT": {
            "keystore": "..\\default.keystore"
        }
    }
}
```

Profiles In the target-specific sections (e.g. *ios*, *android*), you can use *profiles*.

Profiles allow for quick switching between configuration settings at different phases of your development.

For example, you need to use different credentials to sign iOS apps when creating builds for testing and builds for deployment to the App Store.

By creating a different profile in this section, you can quickly change between collections of configuration settings by naming a profile with the `--profile` command-line argument.

If no `--profile` argument is given, Forge attempts to use a profile called `DEFAULT` - it's case sensitive, so you can create and use a profile called `default` if you wish, for example.

Important: When supplying command-line overrides to profile settings, they take a form like `--ios.profile.name value`, where `name` is the setting name to be overidden, and `value` is the setting value.

Available Forge Parameters

general General parameters are configuration settings not related to any particular target.

There is currently only one possible setting in this section to enable *Reloads from external CDNs*:

```
"reload": {
         "external": true
}
```

ios This section contains settings pertaining to building and running Forge apps for iOS.

The device to use when running iOS apps is not profile-specific:

Config Option	Command-line Option	Meaning
device	–ios.device	Either `simulator`, `device` or a specific device ID
simulatorfamily	–ios.simulatorfamily	Either `ipad` or `iphone`
simulatorsdk	–ios.simulatorsdk	E.g. `5.1` or `6.0`

All other settings should be placed inside a *profile*: available settings are shown below:

Profile Config Option	Command-line Option	Meaning
provisioning_profile	–ios.profile.provisioning_profile	Provisioning Profile to embed into your iOS app
developer_certificate	–ios.profile.developer_certificate	Name of certificate to sign iOS app with (OS X only)
devel-oper_certificate_path	–ios.profile.developer_certificate_path	Path to developer certificate (Windows only)
devel-oper_certificate_password	–ios.profile.developer_certificate_password	Password for given developer certificate (Windows only)

For more information about creating provisioning profiles, see *Creating provisioning profiles*.

Note: For more information about building iOS apps on Windows, see *Developing iOS apps on Windows*.

android Use this section for settings relating to building and running Forge apps for Android.

The location of the Android SDK is not profile-specific:

Config Option	Command-line Option	Meaning
sdk	–android.sdk	Path to the Android SDK on your machine.
device	–android.device	Device identifier to run your app on, e.g. `323406C1AD9090EC`
purge	–android.purge	Completely reset all state of the app before running.

The other settings should be in a *profile*:

Profile Config Option	Command-line Option	Meaning
keystore	–android.profile.keystore	Path to your *keystore*
keyalias	–android.profile.keyalias	Alias given to your key in the keystore
storepass	–android.profile.storepass	Password for your keystore
keypass	–android.profile.keypass	Password for your key

We recommend using the command-line switches for `storepass` and `keypass`, rather than placing them in a configuration file, for security reasons.

Internet Explorer Use this section for settings relating to building and packaging Forge apps for Internet Explorer.

Profile Config Option	Command-line Option	Meaning
developer_certificate	–ie.profile.developer_certificate	Filename of your developer certificate (e.g. cert.pfx)
developer_certificate_path	–ie.profile.developer_certificate_path	Path to your developer certificate
devel-oper_certificate_password	–ie.profile.developer_certificate_password	Password for given developer certificate

Using Trigger.io Reload

Trigger.io Reload allows you to update the HTML, CSS and JavaScript in your app for your users without you needing to push an App Store update or your users needing to re-install the app.

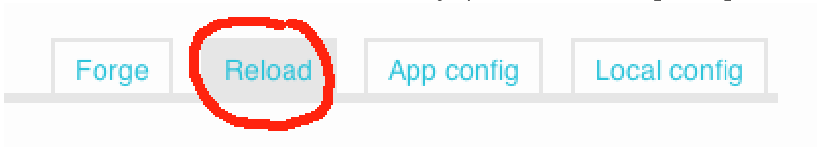
You can push your app content updates via Trigger.io's servers for app sizes up to 250Mb. For larger content, or if you want to host content that is to be re-used in your web app or other apps, then you can *Reload with a 3rd party CDN like Rackspace Cloud Files*.

Read about the *Reload module* to enable it and *learn about the concepts*.

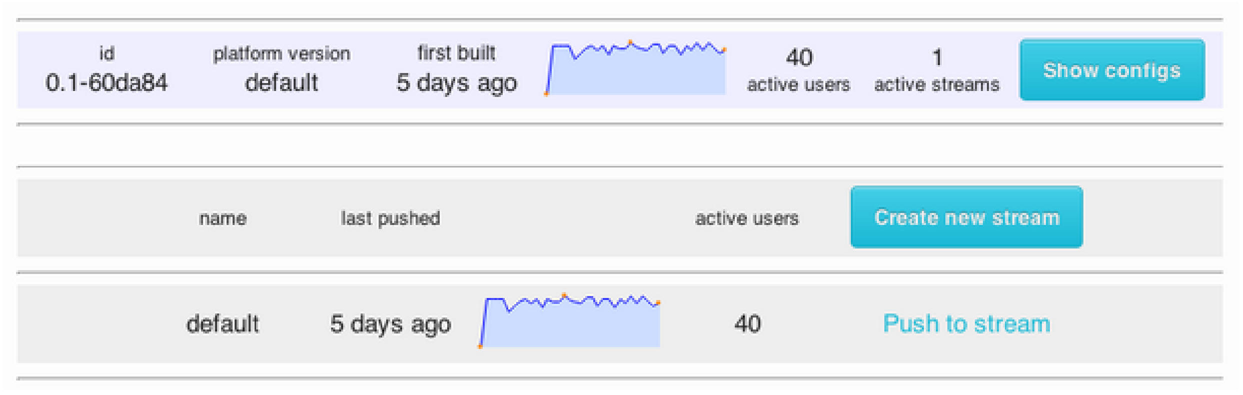
Once you have the module enabled you can push updates to your existing users through the Toolkit UI, command-line tools or *standalone build API*.

How to reload an update with the Toolkit UI

Use the Reload tab in the Toolkit to manage your streams and push updates



When you click on it, you'll see a view something like this (although with less data at first):



id	platform version	first built	40 active users	1 active streams
0.1-60da84	default	5 days ago		

name	last pushed	active users
default	5 days ago	40

In this tab, you can manage your streams of users - this allows you to push different code to different users for A/B testing. By default there is a single stream which enables you to push to all users without any extra configuration.

You can also see your configurations here - how many unique users we have detected that are using your app built against which minor version of the Forge platform.

Once you have built and tested new code that you wish to deploy, you can click the 'Push to stream' link from this view in order to push the update.

Using reload from the command line

Some of the functionality of reload can be accessed via the command line, which allows automation of certain tasks, such as pushing an update. The commands available are:

- `forge reload list`: This will list all available streams for your app
- `forge reload create <streamname>`: This will create a stream called `<streamname>` which can then be pushed to.
- `forge reload push <streamname>`: This will push the last build to `<streamname>`, it is likely you will want to run `forge build reload` directly before this command in order to build your current `src` folder.

Reloading app content from Rackspace Cloud Files or other CDN

Concepts If you are building a content-heavy app including large images or videos it is recommended that you host the app content and Reload it from a CDN like Rackspace Cloud Files.

You must use a CDN with Reload if your app size is greater than 250Mb.

CDNs are optimized for download performance according to your users' locations. Hosting your app content on a CDN with Reload enables you to push large content files down to your app in the background, and link to the same content from your web app or other mobile apps.

Using Reload with a CDN enables you to overcome the challenges of large content in mobile apps. Your app must:

- Be small to allow for fast download onto the device
- Have content local to the app for fast performance and a great offline experience
- Be able to access new content to keep the user engaged

With Reload and a CDN you can have a small app hosted on the App Store or Google Play which downloads large content in the background.

The process When you push a Reload via a CDN the end-user experience is the same and you are able to use the same *Reload APIs* in your app code.

The process for pushing the Reload has some extra steps. You can see these steps in action with our screencast tutorial demonstrating Reload integrated with Rackspace Cloud Files.

The process is:

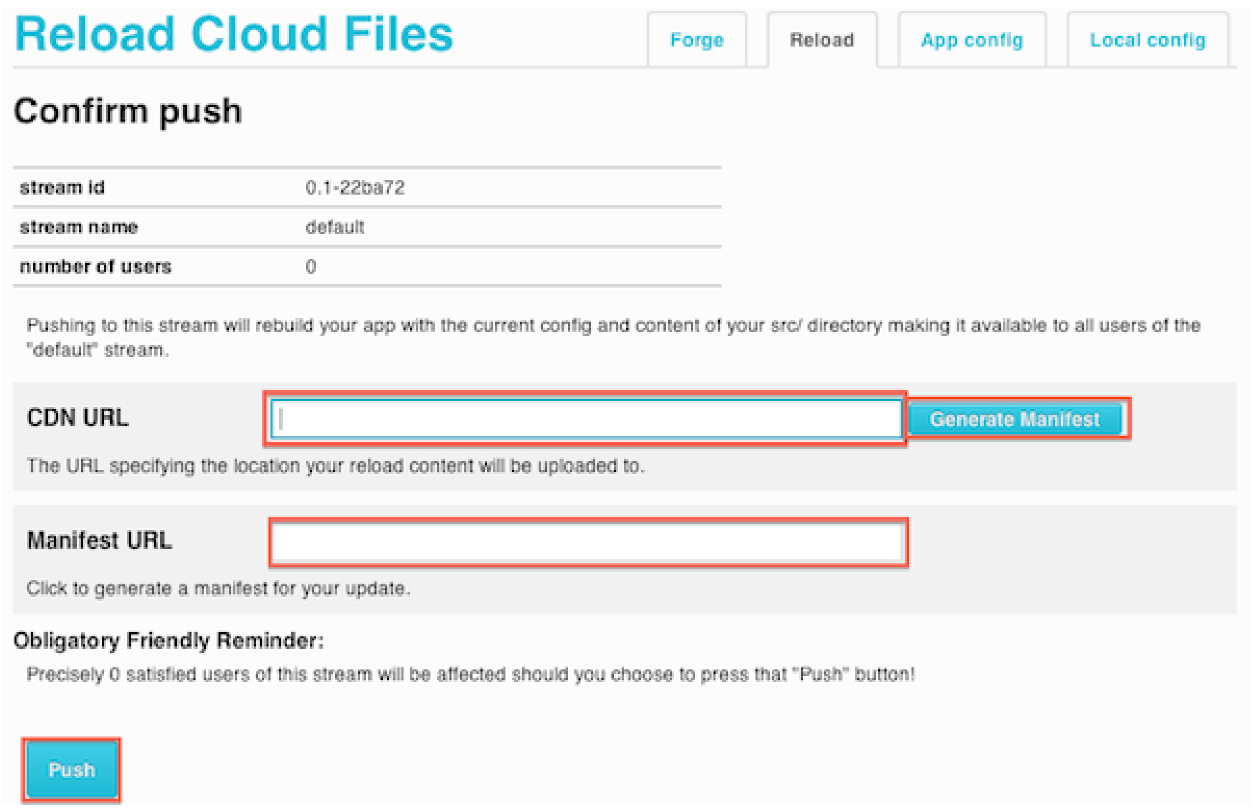
1. *Enable Reloads from a CDN in your Local Config*
2. Create a Reload manifest file
3. Upload the manifest file and files to be reloaded to your CDN and make them publicly available
4. Push your Reload using the Toolkit UI or *standalone build API*

You can use the *standalone build API to push a Reload hosted on a CDN* in which case you must perform steps 2 and 4 manually at the command-line. You can find detailed instructions for those steps by clicking through the links against them above.

But the simplest way is to use the Toolkit UI which automates steps 1 and 4.

How to push a Reload hosted on a CDN via the Toolkit Having *enabled Reloads from a CDN in your Local Config*:

- Go to the Reload tab in your Toolkit, select the config and stream you would like to push to
- Click 'Push to stream' and you will be prompted to provide a CDN URL and Manifest URL



Confirm push

stream id	0.1-22ba72
stream name	default
number of users	0

Pushing to this stream will rebuild your app with the current config and content of your src/ directory making it available to all users of the "default" stream.

CDN URL

The URL specifying the location your reload content will be uploaded to.

Manifest URL

Click to generate a manifest for your update.

Obligatory Friendly Reminder:

Precisely 0 satisfied users of this stream will be affected should you choose to press that "Push" button!

- Once you've entered the CDN URL, you can use 'Generate Manifest' button to automatically create the manifest files and app files in a temporary directory. You should upload all of those files to your CDN.
- Clicking 'Generate Manifest' also populates the Manifest URL field. You must make sure that the url you specified in 'CDN URL' is in fact the root url where your manifest and other files will be hosted.
- Finally, click the 'Push' button to start the Reload

Gotchas

Currently, pushing a Reload update will cause iOS devices to copy files out of the installed app bundle into a secondary area. This is due to sandboxing rules which prevent us directly accessing files in your installed app after a Reload has been completed. In order to avoid your app taking too much storage space on the device, it's recommended you distribute larger files using something like *forge.file.saveURL*, rather than including them in the app bundle.

We are actively looking for a way to avoid this limitation.

Standalone Build API

Although we expect most users to interact with Trigger.io Forge through our command-line tools and browser-based toolkit, we also offer a standlone build API to be used programmatically from other environments.

Using this build API, all of your HTML, CSS and JavaScript is uploaded to our server, along with keys and certificates required for signing purposes. Once the build is complete, the packages can be downloaded for a limited time period.

Due to the nature of this API, builds may take longer to complete when compared to conventional usage. If you are able to use the Trigger tooling, it is recommended to do so.

API details

Package An example form is provided here: https://trigger.io/standalone/package

The various fields are described below:

Name	Label	Expected value
email	email address	The email address you used to sign up to Trigger.io
password	Trigger.io password	The password you gave when you signed up for Trigger.io
src_zip	src folder zip	A zip file containing the contents of your src folder. The zip should not include the src folder itself; i.e. `config.json` should be at the root of the zip file.
and_keystore	Android keystore	The file created by the `keytool` utility. See *Creating a keystore*.
and_storepass	password for Android keystore	The password needed to unlock your keystore.
and_keyalias	alias for Android key	The name of the key in your keystore.
and_keypass	password for Android key	The password for the individual key in your keystore.
ios_certificate	exported iOS certificate	A .p12 file containing your exported iOS developer certificate. On Macs, use Keychain Access; for Windows, see *Creating a signing certificate*.
ios_password	password for iOS certificate	Password you used when exporting your iOS developer certificate.
ios_profile	iOS provisioning_profile	A .mobileprovision file you have downloaded from the iOS provisioning portal.

Usage:

To package your app, use the `/standalone/package` endpoint:

```
> curl \
    --header 'Accept: application/json' \
    --form email=james@trigger.io \
    --form password='my password' \
    --form src_zip=@my_app.zip \
    --form and_keystore=@debug.keystore \
    --form and_storepass=android \
    --form and_keyalias=androiddebugkey \
    --form and_keypass=android \
    -X POST \
    'https://trigger.io/standalone/package'
{"id": "b0a05ec7-1683-40cc-b80b-716ba5d5067a", "result": "ok"}
```

This has started the packaging process and given you an `id` which you can use to track the ongoing processing:

```
> curl \
    --header 'Accept: application/json' \
    --data email=james@trigger.io \
    --data password='my password' \
    -G \
    'https://trigger.io/standalone/track/package/b0a05ec7-1683-40cc-b80b-716ba5d5067a'
{"info": {"output": ""}, "state": "BUILDING", "id": "38b63a52-a35f-49fe-932a-39db3d82951a", "result"
```

At this point, the build has started; repeated calls to `/standalone/track/package` will show when the processing has completed:

```
> curl \
    --header 'Accept: application/json' \
    --data email=james@trigger.io \
    --data password='my password' \
    -G \
    'https://trigger.io/standalone/track/package/b0a05ec7-1683-40cc-b80b-716ba5d5067a'
{"info": {
    "files": {
        "android": "https://trigger.io/media/993100fe3aa844c3ad11575e11aeb9fc/demo-1338404961.apk"
    },
    "output": "[   INFO] Forge tools running at version 3.3.0\n ..."
},
"state": "SUCCESS",
"id": "38b63a52-a35f-49fe-932a-39db3d82951a",
"result": "ok"}
```

We've formatted the last response for ease of viewing: the `state` property transitioning to `SUCCESS` is the key point.

You are able to download the generated files from the URLs specified in the `files` hash.

`Reload` An example form is provided here: https://trigger.io/standalone/reload/push

The various fields are described below:

Name	Label	Expected value
email	email address	The email address you used to sign up to Trigger.io
password	Trigger.io password	The password you gave when you signed up for Trigger.io
project_id	ID of your project	The ID of your project which you can find on your account page
uuid	UUID of your app	The unique identifier for your app which you can see in your app's `src\identity.json` file
stream	stream to push to	The name of the stream e.g. `default`
mani-fest_url	URL to your manifest file	The url to the manifest file which you generated. See the *Reload tools* docs for details
con-fig_json	config.json file	The `src/config.json` file in your app directory

Usage:

To push a Reload to your app, use the `/standalone/reload/push` endpoint.

This is designed to be used with with a *3rd party CDN hosting the files to be Reloaded*:

```
> curl \
    --header 'Accept: application/json' \
    --form email=james@trigger.io \
    --form password='my password' \
    --form project_id='18' \
    --form uuid='362f74be8f4e11e2843012313d00dc45' \
    --form stream='default' \
    --form manifest_url='http://7bda29eaef66b400b7a3-f7a161a32968fd6c080a7ab168500005.r25.cf2.rackcdn
    --form config_json='@src/config.json' \
    -X POST \
    'https://trigger.io/standalone/reload/push'
{"result": "ok", "id": 6583, "manifest": "http://7bda29eaef66b400b7a3-f7a161a32968fd6c080a7ab16850000
```

At this point, the Reload has been pushed successfully and end-users will see the new files being used in their app once they have been downloaded when they next switch focus back to the app.

Working behind a proxy

With the forge command line tool

If you're having trouble running the tools behind a proxy server, you can modify `forge_build.json` in the directory where you installed the forge tools to look like:

```
{
  "main": {
      "server": "https://trigger.io/api/",
      "proxies": {
          "https": "my.proxy.com:8080"
      }
  }
}
```

Note: Make sure to specify the proxy for **https** as all traffic to our services is over https.

With the Trigger Toolkit

As above, you need to modify `forge_build.json` to use a proxy for all https traffic. The location of this depends on your operating system:

On Windows Vista/7 C:\Users\<Your User>\AppData\Local\Trigger Toolkit\build-tools\forge_build.json

On Windows 2000/XP C:\Documents and Settings\<Your User>\Local Settings\Application Data\Trigger Toolkit\build-tools\forge_build.json

On OS X <your home directory>/Applications/TriggerToolkit.app/Contents/MacOS/build-tools/forge_build.json

Using the Python distribution <path to the Trigger Toolkit>/build-tools/forge_build.json

Excluding files from your builds

To exclude files and folders in `src` from being included in the output of `forge build`, you can write a set of exclusion rules in `src/.forgeignore`. The following is an example `.forgeignore` file:

```
ignoreme.txt
*.swp
./tests/
.git/
./identity.json
```

There are two types of exclusion rules: rules that check the *filename* of each file and rules that check the *path* of each file (relative to the `src` directory).

***Excluding files based on their name*:** Any rule without a / symbol in it is a filename rule. If you want to exclude any files called `ignoreme.txt`, the correct pattern to use is just `ignoreme.txt` as shown above.

***Ignoring files with a specific extension*:** Filename rules support glob syntax, so you can have rules that only consider part of the name. For example, `*.swp` will ignore `index.html.swp` and `main.js.swp`.

***Ignoring all folders with a certain name (but not files)*:** To exclude folders that match a rule but not files that match the rule, just add a trailing / symbol. E.g. `.git/` will exclude any folders called `.git` but include individual files with the same name.

***Ignoring a file at a specific path*:** If you want to exclude a file at a particular location in your code, just specify the path to it. E.g. `./identity.json` will exclude `identity.json` at the top level of your `src`, but any other files with that name will be included.

Paths support globs too, so e.g. `./temp/*.js` will ignore all files matching `*.js` inside the temp folder.

Note: Windows users should make sure to always use / symbols as the folder separator (*forward slashes*) in your forgeignore file, these will ensure the exclusion rules work across different platforms

Using hooks

To allow custom steps to be added to the Forge build process the Forge tools support hooks, which are scripts controlled by you which Forge will run at specific points in the build. These scripts are placed in the specific hook folder within a `hooks` folder in the top level of your app (next to `src`). For example placing `hook.py` in `hooks/prebuild` would cause `hook.py` to be executed at the prebuild stage. Hooks are executed in alphabetical order.

Hooks types are defined by the file extension of the hook, the following hook types are currently supported:

- `.py` - Python hooks will be run by executing `python hook.py <target>`.
- `.js` - Node hooks will be run by executing `node hook.js <target>`.
- `.sh` - Shell hooks will be run by executing `./hook.sh <target>` and will not work on Windows.
- `.bat` - Windows batch file hooks will be run by executing `hook.bat <target>` and will only work on Windows.

Important: For `.sh` hooks, you must use a hashbang as the first line, or your hook will fail to execute.

Note: `<target>` will be replaced with the platform currently being built, i.e. `ios` or `android`.

Available hooks are:

- `prebuild` - The prebuild hook executes within the `src` directory before a build is run, however any changes will not affect the `src` directory contents after the build is finished, only the built apps. This allows preprocessing of files to occur.
- `postbuild` - The postbuild hook executes within the the `development` folder once a build has completed, any changes made at this point will affect `forge run` and `forge package` until another build is run.

Examples

Below are a number of example hooks which can be pasted and either used as they are or as a starting point for creating your own.

coffeescript.js

```
// Compile all coffee files in the current tree to js and delete originals

var cs = require("coffee-script");
var fs = require("fs");

files = fs.readdirSync('.');
```

```
var processDir = function (dir) {
    fs.readdirSync(dir).forEach(function (file) {
        var stat = fs.statSync(dir+"/"+file);
        if (stat.isDirectory()) {
            processDir(dir+"/"+file);
        } else {
            var fileParts = file.split(".");
            if (fileParts[fileParts.length-1] == "coffee" && fileParts.length > 1) {
                var script = fs.readFileSync(dir+"/"+file, "utf8");
                fileParts[fileParts.length-1] = "js"
                fs.writeFileSync(dir+"/"+fileParts.join("."), cs.compile(script), "utf8");
                fs.unlinkSync(dir+"/"+file);
                console.log("Compiled "+dir+"/"+file+" to "+dir+"/"+fileParts.join("."));
            }
        }
    });
};

processDir('.');
```

Note: Make you have installed the coffee-script module for node: `npm install coffee-script`

less.js

```
// Compile all less files in the current tree to css and delete originals

var less = require("less");
var fs = require("fs");

files = fs.readdirSync('.');

var processDir = function (dir) {
    fs.readdirSync(dir).forEach(function (file) {
        var stat = fs.statSync(dir+"/"+file);
        if (stat.isDirectory()) {
            processDir(dir+"/"+file);
        } else {
            var fileParts = file.split(".");
            if (fileParts[fileParts.length-1] == "less" && fileParts.length > 1) {
                var script = fs.readFileSync(dir+"/"+file, "utf8");
                fileParts[fileParts.length-1] = "css"
                fs.unlinkSync(dir+"/"+file);
                less.render(script, function (e, css) {
                    fs.writeFileSync(dir+"/"+fileParts.join("."), css, "utf8");
                    console.log("Compiled "+dir+"/"+file+" to "+dir+"/"+fileParts.join("."));
                });
            }
        }
    });
};

processDir('.');
```

Note: Make you have installed the less module for node: `npm install less`

Developing iOS apps on Windows

Forge allows the development of iOS apps on Windows without the use of an OS X machine. To do this you will need a development-enabled physical iOS device and an iOS developer account. In order to sign your application (which is required to install it onto the device, even for testing), we provide a remote signing service, which your app will be sent to, signed and returned as part of the `forge run ios` and `forge package ios` command.

Note: Your iOS device must be enabled for development before our Windows tools can view logging output. This means attaching it to an OS X machine and selecting "Enable for development" in the Xcode organizer.

If you're not able to do this, you can still install apps onto the device, and use the iPhone Configuration Utility to view log output (http://support.apple.com/downloads/#iphone)

Setting up Forge to run iOS apps

Requirements:

- Apple iOS developer account.
- iTunes or iPhone Configuration Utility installed on the machine you are going to develop on
- An iOS device connected via USB to the machine you wish to develop on

In order to sign your application you need to provide us with the following:

- A signing certificate and password
- A provisioning profile

Both of these can be created and managed from the Apple iOS provisioning portal, which should be accessible from the iOS developer center: https://developer.apple.com/ios/. The instructions on that site are for OS X, more detailed instructions for creating a developer certificate on Windows are included below.

Once these are setup you should be able to use `forge run ios` to install the app on your device and see log output in the terminal on your computer.

Creating a signing certificate To create a certificate you need to generate a certificate signing request, in Windows this can be done by following these steps:

- Create a file `request.txt` with the following content, replacing `Connor Dunn` with the name registered to your Apple Developer account:

  ```
  [NewRequest]
  Subject="cn=Connor Dunn,o=User"
  RequestType=pkcs10
  KeyLength=2048
  Exportable=TRUE
  ```

- Run the following command in the same directory as `request.txt`: `certreq -new request.txt`
- On the iOS provisioning portal site choose to create a new certificate and upload the file you just created
- Your certificate request should be approved shortly: when it is, download and open the certificate file. Windows should prompt you to install the certificate.
- Once installed, run the command `certmgr.msc`: this should open a certificate management tool. In this tool browse to `Personal` certificates, you should see the iPhone Developer certificate you just installed.

- You should be able to right click on the certificate and choose All tasks -> Export. Make sure you export the private key as part of the certificate when following the wizard. The password you supply will be the one you need to provide to Forge, and prevents unauthorized users from using the certificate if they were to come into possession of the certificate file.
- You should now be able to configure the `developer_certificate_path` and `developer_certificate_password` in your `local_config.json` file.

Note: See *Configuration for the tools* for more information on your `local_config.json` file.

Creating a provisioning profile Once you have created a certificate you need to create a provisioning profile, this is also done via the iOS provisioning portal website:

- First make sure your device has been added to the provisioning portal, to do this you will need the device identifier (UDID), this can be found by clicking on the device's serial number in iTunes.
- Next create an app id, for development entering `*` as a Bundle Identifier is recommended, as it means multiple apps can be signed with a single provisioning profile.
- Finally create a development provisioning profile, making sure you choose the correct app id and enable any devices you wish to be able to test with.
- You can now download and configure the location of your provisioning profile in `local_config.json`.

Note: Provisioning profiles must be recreated if certificates or devices are changed.

Developing iOS apps on Linux

Forge allows the development of iOS apps on Linux without the use of an OS X machine. To do this you will need a physical iOS device (the iOS simulator will only run on OS X), and an iOS developer account. In order to sign your application (which is required to install it onto the device, even for testing), we provide a remote signing service, which your app will be sent to, signed and returned as part of the `forge run ios` and `forge package ios` command.

Setting up Forge to run iOS apps

Requirements:

- Apple iOS developer account.
- ideviceinstaller installed on the device you are going to develop with
- An iOS device connected via USB to the machine you wish to develop on

In order to sign your application you need to provide us with the following:

- A signing certificate and password
- A provisioning profile

Both of these can be created and managed from the Apple iOS provisioning portal, which should be accessible from the iOS developer center: https://developer.apple.com/ios/. The instructions on that site are for OS X, more detailed instructions for creating a developer certificate on Linux are included below.

Once these are setup you should be able to use `forge run ios` to install the app on your device.

Creating a signing certificate To create a certificate you need to generate a certificate signing request, in Linux this can be done by following these steps:

- Run `openssl req -subj "/CN=Connor Dunn/O=User" -nodes -newkey rsa:2048 -keyout private.key -out request.csr` replacing `Connor Dunn` with your name as registered with your Apple ID.
- On the iOS provisioning portal site choose to create a new certificate and upload the file `request.csr` you just created
- Your certificate request should be approved shortly: when it is, download it to the folder you ran the original command, the next steps assume it is called `ios_development.cer`.
- Run `openssl x509 -inform der -in ios_development.cer -out certificate.pem` to convert the certificate format
- Run `openssl pkcs12 -export -in certificate.pem -inkey private.key -name "iOS Developement Certificate" -out certificate_with_key.p12` to package the certificate and key. The password you supply will be the one you need to provide to Forge, and prevents unauthorized users from using the certificate if they were to come into possession of the certificate file.
- You should now be able to configure the `developer_certificate_path` and `developer_certificate_password` in your `local_config.json` file.

Note: This will leave you with a few files you no longer need, you can safely delete `request.csr`, `ios_development.cer` and `certificate.pem`. You can also delete `private.key` as it is included in the p12 file, if you don't delete it you should store it securely as it can be used to sign with your certificate. You should keep `certificate_with_key.p12` to use with Forge, and make sure it and the password are stored securely so others can't sign as you.

Creating a provisioning profile Once you have created a certificate you need to create a provisioning profile, this is also done via the iOS provisioning portal website:

- First make sure your device has been added to the provisioning portal, to do this you will need the device identifier (UDID), this can be found by clicking on the device's serial number in iTunes.
- Next create an app id, for development entering `*` as a Bundle Identifier is recommended, as it means multiple apps can be signed with a single provisioning profile.
- Finally create a development provisioning profile, making sure you choose the correct app id and enable any devices you wish to be able to test with.
- You can now download and configure the location of your provisioning profile in `local_config.json`.

Note: Provisioning profiles must be recreated if certificates or devices are changed.

2.3.4 Recipes

Using API methods

Most API methods provided by Forge operate asynchronously (exceptions include `forge.is` methods and `forge.tools.UUID`). This means the result of the method is not returned immediately; instead you must provide a function which will be called with the result. Care must therefore be taken when using the methods, as the order in which code is executed can be less obvious.

Here is a simplified example:

```
var url;
forge.tools.getURL("mypage.html", function (myUrl) {
    url = myUrl;
});
alert(url);
```

In this code the alert might be undefined as you cannot be sure the callback would be fired before the next line of code. Instead it would be better to say:

```
var url;
forge.tools.getURL("mypage.html", function (myUrl) {
    url = myUrl;
    alert(url);
});
```

The different callbacks which appear in Forge API methods are described below.

Callbacks

Asynchronous API methods will always have the last two parameters they take as callback functions. The first of which will either be `success` or `callback` and the second will always be `error`. The way these functions are called is described below.

`success` The most commonly used callback is the `success` callback, this is the penultimate parameter to most Forge API methods. The `success` callback can be called with a variety of paramters (including none at all), depending on the particular method.

The `success` callback will only ever be called once, and will only be called if the API method completes successfully.

`callback` The `callback` callback is similar to `success` in that it appears as the penultimate parameter - however, unlike the `success` callback, it may be called multiple times. An example of its use it adding a listener to a button being clicked, as the button can be clicked multiple times the callback may be called multiple times.

`error` The `error` callback is always the final parameter passed to a method. The error callback will always be called with exactly one parameter, which will be an object containing several properties.

The returned object will always contain:

- A `message` property, which is a printable, human readable string describing the error which has occurred.

It will usually also contain:

- A `type` property, this will be one of the following strings and can be used to determine what type of error occurred and how to appropriately deal with it.
- `"BAD_INPUT"` - returned when an API is given invalid parameters, this kind of error should be eliminated before releasing your app.
- `"UNEXPECTED_FAILURE"` - returned when an unexpected error causes the API call to fail, this generally means an error in the forge API and we would be grateful if you could report this kind of error to us.
- `"EXPECTED_FAILURE"` - returned when an expected situation occurs which means the API call is not successful, examples could be a user cancelled action or a timeout. These types of errors should be handled appropriately within your application.

- `"UNAVAILABLE"` - returned when an API method is currently unavailable, this could mean unavailable on the current platform, in the current context, or at this time (i.e. if there is no Internet connection). Your application may also expected some errors of this type. Also returned for non-existant API calls.
- A `subtype` property which will give a more precise description of the error than the `type`, the strings which this may contain are documented with each API method.

It may also contain additional properties which are relevant to the API method, these properties will be documented per method in the API reference.

When errors happen the message property will always be logged by Forge, whether a callback handler exists or not.

JavaScript Libraries

Using JavaScript libraries

It is important to remember that Forge apps are HTML5/JS based and this means they can take advantage of the vast library of Javascript tools and libraries which are freely available. This page lists some of the libraries we find most useful and some tips when using them.

In general (on mobile apps especially) reducing memory usage is important. Using the minimum number of frameworks and ensuring minified versions are always used can improve performance.

jQuery/Zepto

- jQuery is one of the most popular Javascript frameworks and can simplify a number of common Javascript tasks. On mobile however jQuery is a fairly large library and can cause performance issues, and it may be preferable to use Zepto.
- Zepto is a lightweight version of jQuery aimed at mobile browsers. It lacks some less used jQuery features, but is highly mobile optimised and much smaller.
- See http://zeptojs.com/

iScroll

- Creating fixed position or `overflow:  scroll` elements is not possible in the vast majority of mobile browsers currently. iScroll recreates this effect in Javascript.
- Unfortunately this means it can be less efficient than native scrolling would be, so this library should be used with caution. If possible pages which are designed with no fixed header/footer will often run much more smoothly than a more complex page.
- See http://cubiq.org/iscroll-4

Backbone

- Backbone is a lightweight and increasingly popular way of organising your Javascript code. Similar to the concepts of MVC Backbone separates out the sections of your code and provides helpful functions for doing common tasks to do with organising and handling data.
- See http://documentcloud.github.com/backbone/

Example Project: backbone.js, zepto

We've developed a little example project, using a CSS reset, Backbone.js and a couple of pages with transitions.

This project will show you how to:

- include your JavaScript files in your app
- use Backbone.js to present a responsive interface
- use a CSS reset
- implement an example transition between views in your app

Please feel free to base your own projects on this - it's a great springboard for a new project. The code is hosted on github here: https://github.com/trigger-corp/Forge-Bootstrap

Included files

- Backone.js to handle history, user actions, and to structure our JavaScript in general
- HTML5 Boilerplate to reduce the impact of inconsistent rendering defaults on different platforms
- Zepto, a light-weight, mobile-focussed alternative to jQuery, for DOM manipulation

Let's get stuck in To work with your JavaScripts and CSS in the app, just include them in your index.html as you might in a normal website:

```
<link rel="stylesheet" href="css/reset.css">
<link rel="stylesheet" href="css/demo.css">

<script type="text/javascript" src="js/lib/zepto.min.js"></script>
<script type="text/javascript" src="js/lib/underscore-min.js"></script>
<script type="text/javascript" src="js/lib/backbone-min.js"></script>
<script type="text/javascript" src="js/demo.js"></script>
```

Here, we have simply used the HTML5 Boilerplate reset (`reset.css`), JavaScript libraries and two of our own files, `demo.css` and `demo.js`.

It's best to have one entry point for your application, with other included JavaScript files being just libraries, containing functions and objects. When using Backbone, your entry point should set up whatever your app requires to function, then start Backbone's history system.

For example, in this project, we use `$(Demo.init)` to run the following function once, at app startup:

```
// Called once, at app startup
init: function () {
    // Grab the Trigger twitter feed
    forge.request.ajax({
        url: "https://api.twitter.com/1/statuses/user_timeline.json?user_id=14972793",
        dataType: "json",
        success: showIndex
    });

    // to be called once we have the Trigger twitter feed
    function showIndex(data) {
        // Save away initial data
        Demo.items = new Demo.Collections.Items(data);

        // Set up Backbone
        Demo.router = new Demo.Router();
        Backbone.history.start();
    }
}
```

Here we're using the Trigger twitter feed as example data to work with: we use our *request.ajax* function to retreive our tweets, store the data into a collection then start Backbone.

Using Backbone.js `Backbone.history.start()` kicks off Backbone's `window.onhashchange` event subscription. When the fragment of the URL changes, the routes defined in `routes.js` are used:

```
routes: {
    "" : "index",         // entry point: no hash fragment or #
    "item/:item_id":"item"// #item/id
},
```

The routes map a URL to a function. We have two routes defined here: one that matches # (or URLs with no hash fragment) and invokes `index()`, and one that matches `#item/[item_id]`. `item_id` is then passed into `item()` as a parameter. Routes map out the URLs for your whole app.

Using Backbone to manage views inside a Forge app is a great strategy: not only do we build URLs in the history stack (meaning the back button works as expected on Android, for example), we are also able to take complete control over what is displayed in the app, without having to resort to sluggish page loads.

However, especially on mobile platforms, your users will expect some form of dynamic transition from one view to the next; to do that, you can organise your Backbone views into pages.

Page View This snippet shows how we implement pages in this project, with an animated transition as one page becomes active. You can also see us using Zepto for DOM manipulation here.

```
Demo.Views.Page = Backbone.View.extend({
    className: "page",

    initialize: function () {
        this.render();
    },
    show: function () {
        $('.page').css({"position": "absolute"});
        var direction_coefficient = this.options.back? 1 : -1;
        if ($('.page').length) {

            var $old = $('.page').not(this.el);

            //This fix was hard-won, just doing .css(property, '') doesn't work!
            $old.get(0).style["margin-left"] = ""
            $old.get(0).style["-webkit-transform"] = ""

            this.$el.appendTo('body').hide();
            this.$el.show().css({"margin-left": 320 * direction_coefficient});
            this.$el.anim({translate3d: -320 * direction_coefficient +'px,0,0'}, 0.3, 'linear');
            $old.anim({translate3d: -320 * direction_coefficient + 'px,0,0'}, 0.3, 'linear', functio
                $old.remove();
                $('.page').css({"position": "static"});
            });
        } else {
            this.$el.appendTo('body').hide();
            this.$el.show();
        }
        window.scrollTo(0, 0);
    }
});
```

You can `extend()` this page in your own views if you wish, and use the `show()` method to switch from one to another.

For example, in this project, we create a page for the initial view of all the tweets, and a page for each individual tweet when the user selects it.

Using other parts of the Forge API We have already seen the use of `forge.request.ajax` to easily make a request to a remote server. This project makes use of some other Forge APIs too.

In `expand_item()`, we use `forge.tabs.open()` to open an external page new tab in a cross-platform manner. Our documentation for `open()` is *here*.

Lastly, we use `forge.is` in the `click_or_tap()` function so that we can listen for tap events on mobile devices, but click events otherwise. Documentation for easy platform detection can be found here *forge.is.mobile*

```
click_or_tap: function(obj) {
    //for property in obj, add "click " to property and use original value
    var new_obj = {};
    for(var property in obj) {
        if (obj.hasOwnProperty(property)) {
            if (forge.is.mobile()) {
                new_obj["tap " + property] = obj[property];
            }
            else {
                new_obj["click " + property] = obj[property];
            }
        }
    }
    return new_obj
}
```

This is important because the `click` event is less responsive on mobile than `tap`.

That's it Play with the source for yourself, we hope everything is clear.

Still unsure? Want to ask for help? Spotted a mistake in this tutorial? Drop us a line at support@trigger.io and we'll be happy to help.

UI

Dealing with multiple screen sizes

When creating mobile apps it is important to consider that different devices have different resolutions, aspect ratios, pixel densities and potential orientations. This page describes some techniques which may be used to handle these - however none of these remove the need for testing your app on a variety of devices.

The Android developer documentation has a far more detailed document on how multiple screen types are supported by Android: http://developer.android.com/guide/practices/screens_support.html.

Scalable or not An important question to consider when designing your page is whether or not certain parts of the page should be able to dynamically resize (or scale) depending on the screen they are displayed. In some situations the width of an element needs to be fixed, but a paragaph of text may display in a perfectly readable way at several different widths.

If your elements can be resized then it may be as simple as using a relative size when describing them in CSS, such as `width: 80%`. If you required fixed size elements then you may wish to consider detecting the screen size and render content appropriately. A way this can be done using CSS3 is using media queries, which is described in more detail on http://css-tricks.com/6731-css-media-queries/.

Common screen sizes Some of the most common screen sizes to be considered for mobile development are given below. All sizes are given in the number of points (or CSS pixels) which are available to the webview.

- iPhone
- Vertical: 320x460
- Horizontal: 480x300
- iPad
- Vertical: 768x1004
- Horizonal: 1024x748

Glossary Below we define some of the terms used in this page.

pixel The smallest physical square that can be displayed on a screen. Importantly a pixel in css does not always measure a physical pixel used on the device.

aspect ratio The ratio of width to height on a screen. This can vary significantly on mobile devices and also changes when the device is rotated.

pixel density Different devices have different sizes of physical pixels. Android devices are grouped into different desities of pixels, so that different devices can display the same content at a similar physical size. On iOS devices with a retina display have double the pixel density of other devices.

display independent pixel or point Both of these terms are used to describe the "pixel" size used by CSS and HTML rather than the physical pixel size on Android and iOS devices.

Integrations

OAuth with modal views

Modal views are a mobile only feature of Forge.

Modal views allow a second webpage to be loaded "on top" of the main web view in a Forge app. This allows a second page to be displayed without reloading the original, and also provides a dedicated button on the users screen to close the second view when their interaction has ended.

Modal views have two main purposes. The first is to provide quick access to part of your app without breaking the current workflow, and the second is to access an external site to perform a task such as authentication without the chance of the user being "trapped" on the external site if their device has no back button.

Internal pages A simple use case for modal views is to display an options page at any point when users are interacting with a Forge app. This means an options page can be displayed, interacted with and closed, without the current page losing its state. To achieve this simply create a separate options page and when you want to display it use `forge.tabs.open('options.html')` (see *the API documentation*).

External authentication A more advanced use of modal views is to send the user to an external site (potentially your own website), to allow them to authenticate in some way. At this point there are two options, if you control the remote server you can redirect to a `forge:///` url, or whether you control the remote server or not you can watch for a url match pattern and close the modal view when it is found.

Remote redirect In this scenario it is possible to close the modal view and redirect the user to a new local page once the authentication process is complete by redirecting the user to a `forge:///` url. An example of how this could be used would be:

1. Call `forge.tabs.open("https://example.com/login/")`
2. The user logs in to your website
3. Assuming the login is successful your website sets a cookie and redirects the user to `forge:///loggedin.html`
4. The modal view will close automatically, and the original web view will load the local page `loggedin.html`
5. `forge.request.ajax()` requests made by your app will send the cookie your website set (any requests can then be checked against the previous authentication)

Match pattern To watch for a match pattern use code similar to:

```
forge.tabs.openWithOptions({
  url: 'http://my.server.com/login/',
  pattern: 'http://my.server.com/loggedin/*'
}, function (data) {
  forge.logging.log(data.url);
});
```

See *the openWithOptions API documentation*.

See http://developer.chrome.com/extensions/match_patterns.html for documentation on the Match Pattern format and semantics.

In this example the final url in the modal view will be logged, data could also be extracted from it and used to authenticate the user, such as the query string (used for OAuth 1) or the hash fragment (in OAuth 2).

Offline

Referencing local resources

Forge supports building apps that work offline. The HTML, CSS and JavaScript code you write is packaged locally with your app and you can also include image and *media* assets locally.

Here we discuss how to reference your local assets from your code. You can also read about *how to download and cache files* after the initial install.

Relative URLs You should always use relative URLs when referencing files in your `src` directory. For example, with this directory structure for your `src` directory:

```
config.json
identity.json
index.html

css/
    style.css
```

```
img/
    trigger-io-forge-small.png

js/
    jquery-1.7.1.min.js
    main.js
```

You should reference the image from your `index.html` like this:

```
<img id="image" src="img/trigger-io-forge-small.png" />
```

Similarly, you could set it as background image for an element by adding this to your `style.css`:

```
#background {
        height: 46px;
        background: url('../img/trigger-io-forge-small.png');
}
```

We've used an image in our examples here but you can reference other assets from the HTML (like JavaScript styles and CSS stylesheets) in the same way.

Caching files

File caching a mobile only feature of Forge.

File caching allows you to download a remote resource (such as an image), and keep a reference to the local file. These can then be used in the future to display to the user, even if no internet connection is available. Importantly it is possible to save Forge file objects as Forge preferences which will be persisted between app launches.

Example This is a quick example of how we can store the logo to our site locally without having to include it in our app when we build it. In this example we download the Forge logo from the Trigger.io site: https://trigger.io/forge-static/css/trigger/f/logo-header-small-forge.png.

Downloading and storing file reference First we need to be able to cache the file and save the file object to use later.

Example:

```
var cacheLogo = function () {
  forge.file.cacheURL("https://trigger.io/forge-static/css/trigger/f/logo-header-small-forge.png", fu
    // File cached save the file object for later
    forge.prefs.set("logo-cache", file, function () {
      // Logo saved
      showCachedLogo(file);
    });
  });
}
```

Checking cached file Cached files may be deleted by the operating system to free up space, you should always check a cached file is still available before using it.

Example:

```
var checkCachedLogo = function () {
  forge.prefs.get("logo-cache", function (file) {
    if (!file) {
      // Logo never cached
      cacheLogo();
      return;
    }
    forge.file.isFile(file, function (isFile) {
      if (!isFile) {
        // File no longer available
        cacheLogo();
      } else {
        // Logo available
        showCachedLogo(file);
      }
    });
  });
}
```

Using a cached file Finally we need to be able to display a cached file, to do that we can use the following code:

Example:

```
var showCachedLogo = function (file) {
  forge.file.URL(file, function (url) {
    var logo = document.createElement('img');
    logo.src = url;
    document.body.appendChild(logo);
  });
}
```

Complete example Putting those functions together we get an example which will cache a logo and display it from the cache:

Example:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <h1>Hello caching world!</h1>
    <script>
      var showCachedLogo = function (file) {
        forge.file.URL(file, function (url) {
          var logo = document.createElement('img');
          logo.src = url;
          document.body.appendChild(logo);
        });
      }

      var cacheLogo = function () {
        forge.file.cacheURL("https://trigger.io/forge-static/css/trigger/f/logo-header-small-forge.p
          // File cached save the file object for later
          forge.prefs.set("logo-cache", file, function () {
            // Logo saved
            showCachedLogo(file);
          });
        });
      }

      var checkCachedLogo = function () {
        forge.prefs.get("logo-cache", function (file) {
          if (!file) {
            // Logo never cached
            cacheLogo();
            return;
          }
          forge.file.isFile(file, function (isFile) {
            if (!isFile) {
              // File no longer available
              cacheLogo();
            } else {
              // Logo available
              showCachedLogo(file);
            }
          });
        });
      }

      checkCachedLogo();
    </script>
  </body>
</html>
```

Preparing your app for release

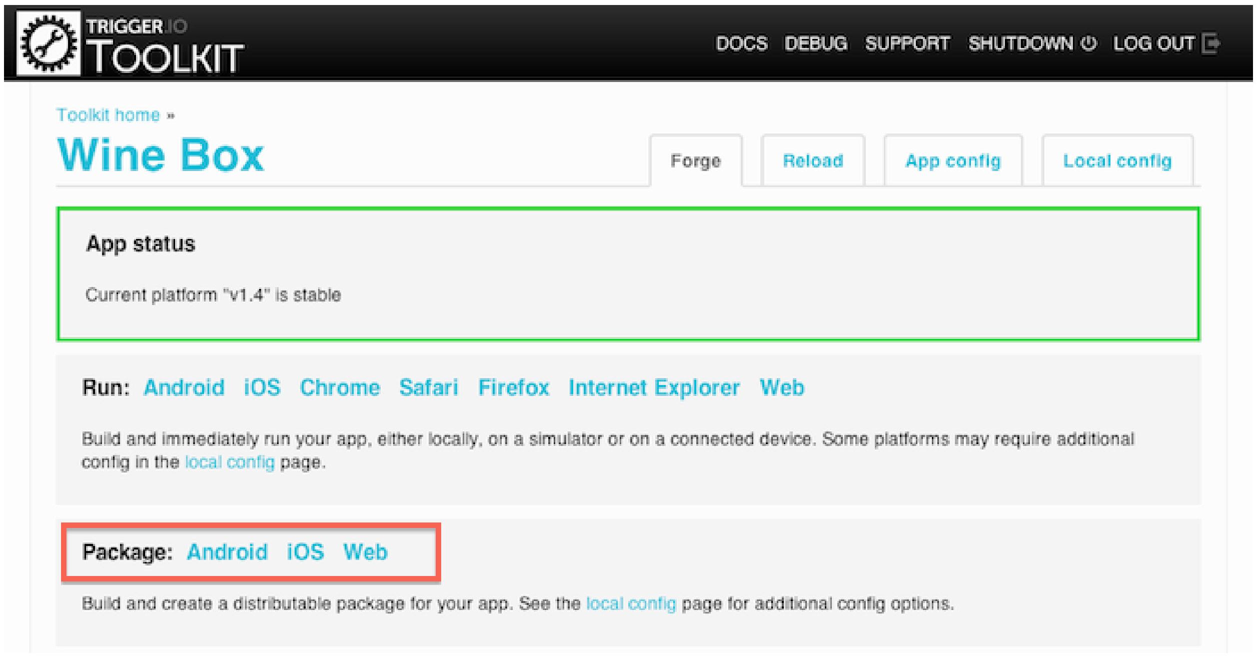
Releasing your mobile apps

After you have finished building and testing your app, the next step is to prepare packaged output ready for submission to the app store of your choice.

We help you with this process on the Android and iOS platforms with the `forge package` command at the command-line or through the Toolkit UI.

Updating existing apps If you are using Forge to update an existing app which was not built on the platform, you will need to specify the package name manually, to match the one you had in your old app. To do this, see *package_names* in the configuration documentation.

Android For Android, create an APK from the Toolkit by using the package link on your app home screen:



Wine Box

App status

Current platform "v1.4" is stable

Run: Android iOS Chrome Safari Firefox Internet Explorer Web

Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

Package: Android iOS Web

Build and create a distributable package for your app. See the local config page for additional config options.

If you prefer the command-line tools, use `forge package android`: add the `--help` switch to see all available command line options.

To package Android apps, you need to have created a "keystore" with which to sign your app. You should keep this keystore safe, and observe the normal password safety precautions to prevent others from being to update your own apps.

Creating a keystore Full instructions on creating a keystore are available here: http://developer.android.com/guide/publishing/app-signing.html, in the "Obtain a suitable private key" section.

You will need the `keytool` command, which is provided by Java.

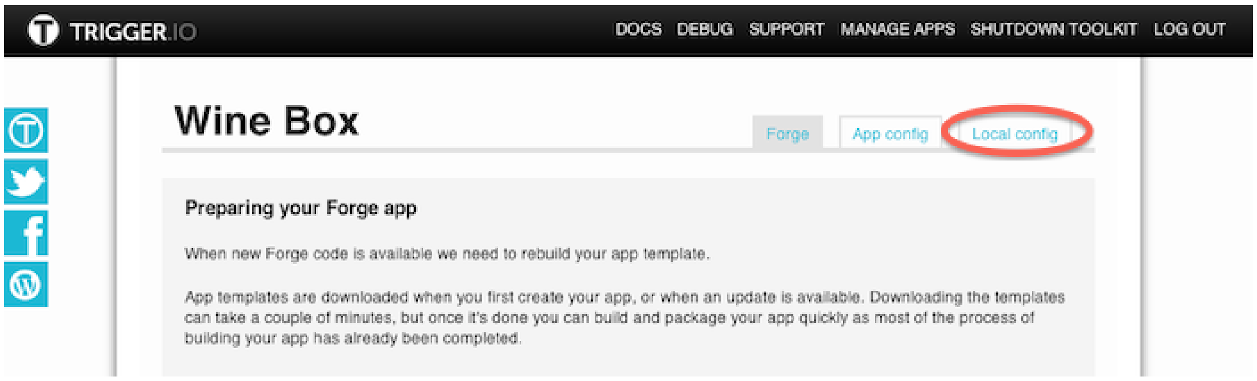
For example, you might use a command something like this:

```
keytool -genkey -alias MY_KEY_NAME \
  -keypass MY_KEY_PASSWORD -validity 20000 \
  -keystore my_keystore.keystore \
  -storepass MY_KEYSTORE_PASSWORD
```

Here, the uppercase parameters should be replaced with passwords and names of your choosing; the passwords can be the same, if you wish.

That command will create a file called `my_keystore.keystore` in the current directory, which we will use in the next step.

Creating an APK After you have created your keystore, you need to enter the details in the Local Config tab in your Toolkit UI:



Preparing your Forge app

When new Forge code is available we need to rebuild your app template.

App templates are downloaded when you first create your app, or when an update is available. Downloading the templates can take a couple of minutes, but once it's done you can build and package your app quickly as most of the process of building your app has already been completed.

Or, if you are working at the command-line, to use the keystore we created in the previous step, we would run something like this:

```
forge package android \
  --android.profile.keystore my_keystore.keystore \
  --android.profile.storepass MY_KEYSTORE_PASSWORD \
  --android.profile.keypass MY_KEY_PASSWORD \
  --android.profile.keyalias MY_KEY_NAME
```

Note: For a convenient way to specify and these options in a file, see *Configuration for the tools*.

A `release` directory has now been created; you will find your release-ready APK file in the `android` sub-directory.

iOS For the iOS devices, creating a packaged IPA file serves two purposes:

1. creating an installable IPA which you can run on your own devices, or send to testers for them to test.
2. creating a IPA which is ready to submit to the App Store.

In the first situation, you need to create and use a development provisioning profile, with all the devices for which the IPA is enabled specified in it.

For the second situation, you need to create and use a release provisioning profile, and your IPA won't run directly on any devices: it is only useful for submission to the App Store.

Therefore, our recommended approach is to use a development wildcard provisioning profile (one that works for any of your apps), right until you're ready to release to the App Store. Only at that point should you switch to your release provisioning profile.

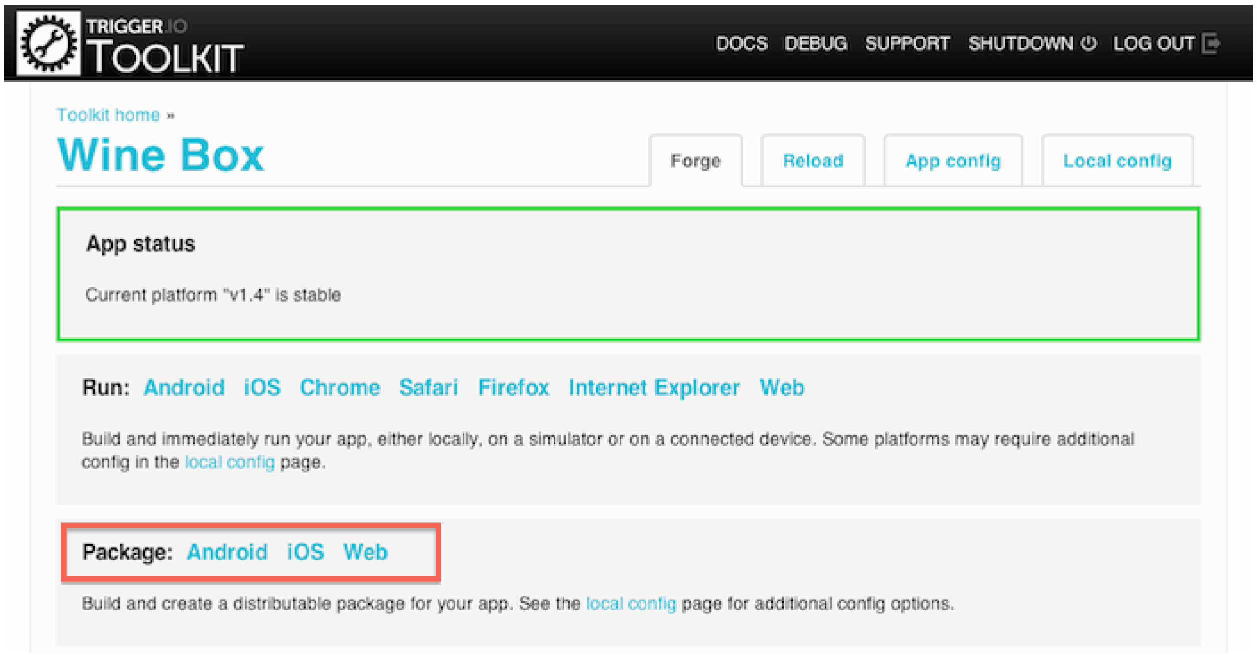
Creating provisioning profiles The Apple developer center has good documentation about creating provisioning profiles, developer certificates and app IDs that should be your main reference.

When working with Forge, you will want to use a development wildcard provisioning profile when in development and testing phases, and a release provisioning profile when submitting to the App Store.

The result of these instructions assume you have your two provisioning profiles stored in the current directory, called `Development.mobileprovision` and `Release.mobileprovision` respectively.

Creating an IPA After you have created your provisioning profile, use `forge package ios` to produce an IPA file.

You can run this command through the Toolkit UI by using the package link on your app home screen:



Wine Box

App status

Current platform "v1.4" is stable

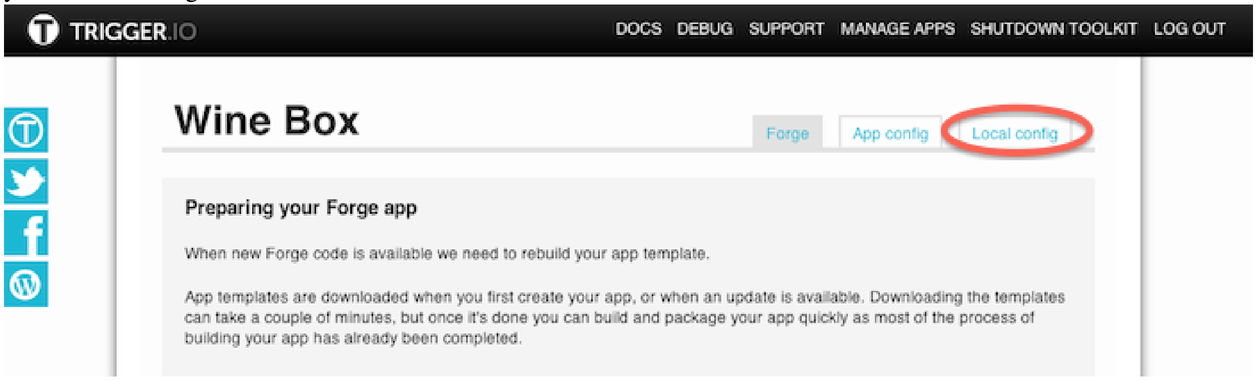
Run: Android iOS Chrome Safari Firefox Internet Explorer Web

Build and immediately run your app, either locally, on a simulator or on a connected device. Some platforms may require additional config in the local config page.

Package: Android iOS Web

Build and create a distributable package for your app. See the local config page for additional config options.

Before packaging, you will need to specify your provisioning profile. And depending on your local settings, you may also need to know the exact name of your developer certificate: more on that below. In the Toolkit UI do this using your Local Config tab:



Wine Box

Preparing your Forge app

When new Forge code is available we need to rebuild your app template.

App templates are downloaded when you first create your app, or when an update is available. Downloading the templates can take a couple of minutes, but once it's done you can build and package your app quickly as most of the process of building your app has already been completed.

At the command-line, assuming you have a provisioning profile called `Development.mobileprovision` in the current directory, you would use a command like:

```
forge package ios --ios.profile.provisioning_profile Development.mobileprovision
```

The `forge package ios` prints out some useful configuration data as it runs, such as the devices this IPA will work with whether you used a Release provisioning profile and the app ID.

In the `release` directory, there will now be an `ios` sub-directory, containing your IPA.

Getting the IPA onto your device If you used a development key, you can now use iTunes to install the IPA onto your iPhone or iPod:

- drag the IPA onto the "Library" section in iTunes
- drag the app from the "Apps" section of iTunes onto your connected device

Submitting the IPA to the App Store If you've used a Distribution (App Store) provisioning profile to create your IPA, you can submit that IPA to the App Store for review.

Refer to Apple's Documentation for how to do that.

You will need a Mac computer to complete this submission, because you must use Application Loader (a Mac-only program) to upload the IPA.

Common problems If you have more than one developer certificate on your machine, you may hit errors like:

```
[  ERROR] Something went wrong that we didn't expect:
[  ERROR] Failed when running /usr/bin/codesign
```

Running the `forge package ios` command again with the `-v` flag for verbose output gives more information:

```
[  DEBUG] Running: ('/usr/bin/codesign', '--force', '--preserve-metadata',
  '--entitlements', '/Users/james/../.template/generate_dynamic/dev.entitlements',
  '--sign', 'iPhone Developer', '--resource-rules=/myapp.app/ResourceRules.plist',
  '/myapp.app/')
[  DEBUG] iPhone Developer: ambiguous (matches
  "iPhone Developer: James Brady (5W89HYT9F3)" and
  "iPhone Developer: James Brady (A639RL926N)" in
  /Users/james/Library/Keychains/login.keychain)
```

Here, there are two developer certificates for "James Brady" on the machine, and we have to specify the exact certificate to use with:

```
forge package ios --ios.profile.provisioning_profile Development.mobileprovision \
  --certificate "iPhone Developer: James Brady (5W89HYT9F3)"
```

If you encounter errors about a mismatched profile ID, e.g.:

```
[ ERROR] Provisioning profile and application ID do not match Provisioning
profile ID: A8N4D63NB6.io.trigger.forge7faf8ebcb8a111e1910212313d1adcbe
Application ID: A8N4D63NB6.com.spiffyapp Please see "Preparing your apps
for app stores" in our docs: http://current-docs.trigger.io/releasing.html#ios
```

This is because when you created your provisioning profile, you didn't use the ID automatically generated by Trigger (`io.trigger.forge7faf8ebcb8a111e1910212313d1adcbe`) in this case.

This is no problem: just update your `config.json` to override the package name to match your provisioning profile. In this example, you'd include:

```
"package_names": {
    "ios": "com.spiffyapp"
}
```

For more information, see *package_names: Built package names*.

Wireless distribution on iOS

If you are using Adhoc or Enterprise distribution certificates when packaging your iOS app, you are able to wirelessly distribute the resulting IPA file. To help you do this, a wireless distribution plist template will be generated and output alongside your IPA file in the `release/ios` folder.

In order to use this plist you will need to modify 3 URLs: all URLs must be absolute and point to a downloadable file. Once you have modified these URLs you can host the plist file and access it from the device to install the app.

The URLs that need to be modified in the plist are:

- `http://www.example.com/app.ipa`: is a download URL for the IPA just generated
- `http://www.example.com/image.57x57.png`: is a 57x57 pixel PNG icon for your app which will be displayed while it downloads
- `http://www.example.com/image.512x512.jpg`: is a high resolution image for your app

More information on wireless app distribution can be found in Apple's documentation: http://developer.apple.com/library/ios/#featuredarticles/FA_Wireless_Enterprise_App_Distribution/Introduction/Introduction.html

Releasing your browser add-ons

After you have finished using `forge build` and `forge run` to create your app, the next step is to prepare packaged output ready for submission to the app store of your choice.

The process for this varies according to the browser platform.

Chrome Use the `chrome://chrome/extensions` page in your Chrome browser to package your app as a `.crx` file.

Pre-reqs

- *Load the extension into your Chrome browser*

How to create the `.crx` Follow these instructions in the Chrome documentation to package your app.

Important: you must keep the `.pem` file that is generated by chrome in the same directory as the `chrome.crx` package so that you can sign future builds with the same key to push updates to existing users

Firefox

Pre-reqs

- Firefox is a premium platform, so you will need to sign-up for a paid account. Contact us to upgrade.

How to create the `.xpi` Simple zip the contents of the *firefox/development* directory as a `.xpi` file to create a Firefox installer package:

```
cd development/firefox
zip addon.xpi -r *
```

Safari

Pre-reqs

- Safari is a premium platform, so you will need to sign-up for a paid account. Contact us to upgrade.
- Create an Apple Developer Account and sign-in to the Member Center
- Sign-up for the Safari Developer Program and access the Safari Dev Center

- Go to the Developer Certificate Utility and create a Safari extension certificate by following the instructions having clicked 'Add Certificate'
- Download the certificate you just created and add it to your Keychain

How to create the `.safariextz` First you need to enable your browser for extension development - do this by enabling the developer tools using these instructions.

Once you have opened up the Extension Builder in Safari, click the '+' button in the bottom left and select the `development/forge.safariextension` sub-directory of your app directory.

Then click the 'Build Package...' button to create your `.safariextz` package.

Internet Explorer

Pre-reqs

- IE is a premium platform, so you will need to sign-up for a paid account. Contact us to upgrade.
- Install NSIS

How to create the `.exe` Simply run:

```
forge build ie
```

This will create a `.exe` for 32-bit and 64-bit platforms in the `release/ie` directory. The 64-bit installer installs the add-on in both the 64-bit and 32-bit versions of IE.

Contact us for information on how to customize the installer package.

How to sign a release First you need to obtain a Windows Code Signing Certificate. Most certificate vendors can supply this but Microsoft also maintain a list at: http://social.technet.microsoft.com/wiki/contents/articles/2592.aspx

You will also need access to Microsoft's `signtool.exe` binary. This is installed by all versions of Microsoft Visual Studio including the free Express version which can be obtained from: http://www.microsoft.com/visualstudio/eng/downloads

Once you have a certificate you can perform the following step to sign your add-on and the installers:

```
forge build ie --ie.profile.developer_certificate "your_certificate_filename.pfx" \
               --ie.profile.developer_certificate_path "c:\path\to\your\certificate" \
               --ie.profile.developer_certificate_password "your_certificate_password"
```