FAIMS PROJECT

# FAIMS 3 ELABORATION REPORT

DESIGN TECHNOLOGIES FOR 2021-2022
THE MOBILE APP DEVELOPMENT CYCLE

| | |
|---|---|
| For | Technical Advisory Group & Steering Committee |
| License | Creative Commons 4.0 International - With Attribution |
| Contact | Brian Ballsun-Stanton <brian@faims.edu.au> |
| Version | 1.0.0-SC |
| DOI | 10.5281/zenodo.4616766 |
| Authors | Brian Ballsun-Stanton<br>Rini Angreani<br>Steve Cassidy<br>Simon O'Toole<br>Nuria Lorente<br>Elizabeth Mannering |

18 March 2021

MACQUARIE University SYDNEY·AUSTRALIA

ARDC Australian Research Data Commons

CSIRO

UNSW SYDNEY

LA TROBE UNIVERSITY

AARHUS UNIVERSITY

aarnet Australia's Academic and Research Network

## Changelog

| | | |
|---|---|---|
| 1.0.0-SC | 2021-02-25 | Typesetting in LaTeX |
| 0.9.0-SC | 2021-02-18 | Report with comments to Shawn Ross for advancement to steering committee. |
| 0.2.0-TAG | 2021-02-12 | Adding feedback from emails |
| 0.1.0-TAG | 2020-12-17 | Final typesetting and release to the Technical Advisory Group |
| 0.0.5 | 2020-12-16 | Edit by BBS |
| 0.0.4 | 2020-12-15 | Edit by Shawn Ross (versioned named before edit) |
| 0.0.3 | 2020-12-09 | First edit by BBS |
| 0.0.2 | 2020-12-03 | Release to dev team, Update QA work division wording, Updates on framework/platform choices, SC and BBS discussed frameworks and JSON Forms choices. |
| 0.0.1 | 2020-11-26 | Initial composition |

Macquarie University NSW 2109

Australia

E: info@faims.edu.au

W: faims.edu.au

ABN 90 952 801 237

CRICOS Provider 00002J

# Contents

# 1 Design objectives

FAIMS3 is a ground-up rewrite of the FAIMS Mobile (v2.6) offline-capable, geospatial, multimedia, field-data collection application (Ballsun-Stanton et al. 2018). This rewrite is designed to be multi-platform, maintainable, and to support data collection at a citizen-science scale. The code and platform should last for at least five years, assuming regular maintenance.

These are difficult objectives.

Our memorandum of understanding with the developers lists the following intended capacities:

- Replicate all FAIMS v2.6 features that are currently used as part of three projects' research practice: CSIRO environmental geochemistry, LTU Mungo Lakes archaeology, and UNSW Oral History.
- Allow self-service customisation and deployment via a web application without needing FAIMS team intervention for the vast majority of deployments.
- Operate cross-platform, running the same code on Android, iOS, and 'desktop'.
- Allow data 'round trip' to web and desktop applications. Round-trip, aspirationally, can be considered as data captured in-field, edited externally, then returned to the device in editable format.
- Improved scalability and performance. Our target is ten times the number of records per deployment compared to v2.6 plus server-to-server synchronisation.

Pragmatically, we had to balance these objectives against available skills, the costs of retraining/hiring, development budgets/timelines, and future maintainability. During elaboration, we had several objectives:

1. Try to quickly falsify technologies to rule-out unsuitable ones;
2. Build a team – determine patterns of communication, management, and capabilities;
3. Explore where we needed to take on technical debt, what developmental affordances were supported by our choices, what external dependencies will be a worthwhile tradeoff between time saved versus external support risk, and where we will have to rebuild components because external support was insufficient;
4. Explore the technologies needed to support development and make a persuasive case for DevOps choices;
5. Produce elaboration outputs that indicate the feasibility of our desired development; and
6. Elicit feedback from the project community to provide an external perspective that can catch errors and suggest improvements.

# 2 Technologies

## 2.1 Programming language and datastore

Language-wise, our primary inquiry compared the modern Javascript app development ecosystem versus promoted languages like Google's Dart. Javascript has advanced significantly over the years, moving from a browser-based web-scripting language to a modern, fully fledged programming language capable of managing front-end experience and back-end business logic. Dart, on the other hand, is a recently designed, purpose-built language from Google that assessors found persuasive in the FAIMS submission to the DataApp Challenge competition as a 'well-supported native-code compilation framework' (Bureau of Reclamation 2017). While Dart compiles to Javascript or native-code application, none of the proposed programming team has experience with the language.

Initial inspection, unfortunately, was enough to discount Dart. While Node.JS has excellent *offline* NoSQL support with Apache's PouchDB, and server-side synchronisation with CouchDB, no similarly mature packages exist for Dart. We also found no persuasive mapping packages. Dart is certainly a viable tool for standard web applications built using a modern client-server or online serverless environment. However, its design priorities and library support do not emphasise multi-OS, geospatial, offline capabilities. Furthermore, Dart's library environment is less mature than Node.JS. This lack-of-maturity for offline and geospatial support, combined with Google's proclivity to abandon underperforming projects, argued against Dart.

We also had to decide between writing native applications versus making a pure Javascript webapp. Native applications in Java, Objective-C, and a language appropriate for desktop provide lower-level access to operating systems, sensors, and device capabilities in exchange for a tripled programming load. At the other end of the spectrum, we could pursue a pure-browser progressive web application approach, offering a seamless multi-platform experience in the browser. We chose neither of these routes. On one hand, we could not deploy a pure progressive web application that runs only in a browser since we had to access device capabilities that are carefully and appropriately isolated from the browser. On the other, we chose not to deploy a purely native application due to the expense of development without a compelling need for the performance gains offered by low-level operating system access.

We chose the middle route: wrapping a progressive Javascript 'single page application' in native code. That way, we can 'deploy an app' (native code) to all platforms (AppStore, Google Play, Microsoft Store) but written in Javascript and shipped with enough webbrowser-like code so that it acts and appears as a normal application on someone's mobile or desktop device. This approach, deploying 'native progressive webapps' is an extremely common programming practice for those projects needing to deliver a multiplatform experience without the ability to support three or more distinct programming teams.

### 2.1.1 Programming language

We will use ECMAScript 6, known as Javascript, with JSX code extensions. This choice was primarily motivated by the React framework and ecosystem – though we are still investigating the need for a framework like React at all. We will be using Node.JS v14 for Alpha development as it is the current LTS release. It will reach end-of-life on 2023-04-30. If possible, we will try a version upgrade to v16 as early as possible to extend EOL to 2024. We recognise that these short 'long-term service' releases will impose substantial maintenance – but as we plan for a yearly maintenance cycle to allow testing and ensure support for major Android and Apple releases, bumping and testing the node version will be part of that plan. Furthermore, we anticipate a need for more frequent security releases due to the enormous complexity of the Node.JS package ecosystem.

All base components of FAIMS3 developed by us will be written in Javascript: the application, the module designer, and whatever supporting server infrastructure is required. There may also be a need for OS specific plugins to support specific functionality, though we will try to avoid this exigency whenever possible.

### 2.1.2 Datastore

We have chosen to use the CouchDB document store as our fundamental database. It is a robust multi-master document store designed to deal with intermittent connectivity and opportunistic synchronisation. It is ACID-compliant and supports integral document versioning implying support for a version-history similar to that of FAIMS2.6. It also has a stronger focus on mobile application support via PouchDB. Finally, the Apache License used for CouchDB and PouchDB is more permissive than the viral AGPL. We have therefore chosen the robust and mature pair of CouchDB 3 / PouchDB 7 as the FAIMS3 datastore. Elaboration experiments with these DBMSes in React and React-Native have proven satisfactory. We hope to be using GeoCouch as a spatial index addon for CouchDB and GeoPouch as its local complement but both GeoCouch and GeoPouch are aging projects that do not have current updates. While there is legitimate concern around GIS performance, we have not investigated implementation options for the specific geospatial functionality and tradeoffs needed for our target users and their modules.

## 2.2 Framework, essential packages, and toolchain

### 2.2.1 Framework

We are not yet convinced that we want to take on the technical debt of the React framework. However, we have investigated React and React Native, and we undertook preliminary investigations of Ionic and Vue. As a result, if we do use a framework, it will be React 17. React 17 supports gradual upgrades and, considering the current developer-base of the framework, it will still be in use in 2025.

The choice of React is driven by the experience the team has with it and the lack of clear differentiators with Vue. We did not investigate Angular due to the expert advice of our programmer who is involved in a complex and entailed migration away from Angular.

The main alternative to using React would be to use a collection of more limited frameworks for specific tasks such as building web components, producing page layout, etc. The main advantage of this approach would be reduction of overall project dependencies, but we currently believe that the value of React outweighs its concomitant dependency risks.

One of our elaboration exercises was exploring React Native. React Native did not present the same capabilities as React in terms of its supported packages and is not yet at a stable release. Due to its relative immaturity and lack of compelling advantages over React, we chose not to explore React Native further.

We also chose not to explore the Ionic framework. While it does have some useful upgrades for Capacitor, discussed in the next section, the features offered in either its open source or enterprise versions were not required during elaboration. During development, however, if one of those Ionic plugins for Capacitor are needed, we would need to make a judgement call about including the whole Ionic framework.

### 2.2.2 Packages and native runtimes

#### Compiling to Native runtimes

Our cross-platform commitment requires systems able to compile to Apple and Android native runtimes, as well as building to Electron's native desktop wrappers and a Progressive Web Application for pure web browser support. We investigated Apache Cordova, Ionic's Capacitor, and Facebook's React Native for this purpose. React Native did not offer compelling advantages for our use-case, and increased development friction. React Native's plugin ecosystem also appears weaker compared to the first-class support from Ionic. While Cordova *nee* PhoneGap was initially our preferred native runtime, it does not offer the same access to web APIs and native platform capabilities.

Ionic's open source offering in this space is Capacitor. Capacitor, billed as the successor to Cordova, compiles to a Native Progressive Web App, giving us the flexibility to include native code for OS-specific features. Capacitor offers critical capacities required for feature delivery and performance. Capacitor compiles its Javascript into platform-specific IDE builds which allow us to 'add custom native code...without having to build a new plugin for it [and] provides better app maintainability as new mobile operating system versions are released'. This support for mobile operating system upgrade and migration is critical to FAIMS3's long-term sustainability and is a major benefit over Cordova for this project.

We investigated Ionic Enterprise due to their claims of 'enterprise security updates' and its Capacitor enterprise plugins. We have, however, seen no compelling features in its enterprise support. If we decide that JSON Forms, when implemented in detail, do not provide the schema support we require, we will investigate Ionic again, since it offers some additional functionality in that regard.

### JSON Forms

We are provisionally satisfied with the functionality of JSON Forms, which renders a JSON Schema as a dynamic form. Although we have some concerns about its capacity to accommodate the complexity we will ask of the system, we have not found a persuasive alternative for dynamic form rendering based on a JSON schema. If JSON Forms fails, we will need to contribute to the package, fork it, or build our own. This will be a significant but necessary expense, but one that would result in downscoping other features.

The main shortcoming of JSON Forms arises from the JSON Schema standard itself, which does not allow sub-typing (defining types as specialisations of other types). This lack of subtypes makes it hard to extend the basic set of types to cover new kinds of input fields. For example, we would like to have a library of predefined types like 'GPS Location' that could be used in any application. Two things stop us. First, we cannot build a 'standard library' of these custom subtypes in the style of the FAIMS v2.6 autogenerator. Each JSON Schema would need to be entirely self-contained. Secondly, even if we declared these subtypes in a schema, the declaration is not of an abstract type but of a full form element. Having two or more GPS Locations in one form would either require them to have the same 'prompt' (form label) or be defined as separate types (top-left-GPS-location, secondary-GPS-location, etc). This constraint on custom fields means that we will need to extend the JSON Schema to allow us to write appropriate specifications. Such development would also prevent use of any existing JSON Form library in our application.

The JSON Schema is basically sound, but we need to build our own version that meets project requirements. The architecture of the JSON Forms library is quite complex, as it handles multiple host frameworks and multiple output rendering options. We can learn from and build upon the existing library but we will need our own implementation. A major effort of alpha development will be to design and build a sub-system that can instantiate appropriate forms from a JSON representation.

### Leaflet

The other major avenue of investigation during elaboration was mapping / mobile GIS support. We currently favour an approach using Leaflet, a Javascript tile-render. Leaflet supports many of the features researchers used in FAIMS v2.x – raster maps, vector maps, dynamic points, vectors, and rudimentary vector styling. It also utilises hardware acceleration on mobile devices, which should result in a higher performance. We are investigating multiple offline modes for Leaflet, either with a global caching functionality or something like Leaflet Offline, another tradeoff between development time and external dependencies. Since Leaflet is an open-source javascript project, however, it does not present the same external library trap as Nutiteq (a proprietary mobile GIS) did for FAIMS v2.x. Nutiteq locked us into a specific Android API and Sqlite version and then stopped supporting their code. Because Leaflet does not dictate our datastore or ship with native code that we have to incorporate into our application, it is much easier to switch it out later.

Geospatial storage has advanced significantly since our last elaboration in 2013 (when the only viable option was Spatialite extensions to SQLite rendered via a mobile GIS). We will store GeoJSON in our CouchDB document store. Hopefully, as discussed earlier, using the GeoCouch spatial extension to allow for spatial queries to reduce rendering complexity. We do not anticipate building significant GIS query capabilities into our fundamental datastore as those expensive features were underutilised by clients in FAIMS v1.0-2.6. Instead, we will rely on building that functionality into plugins when explicitly required (and funded) by clients. A more modular design will also allow us to replace components as they reach end-of-life without impacting the rest of the application.

However, other members of the FAIMS Leadership team have experience with Leaflet and geojson and note that both have a ceiling when it comes to fast display of complex and large (10000+) geospatial datasets. We do not plan to implement significant GIS functionality in the application, but being able to visualise collected data is an essential requirement. Therefore, it is important that users be able to visualise all the points they have collected on an open street map or satellite view as a minimum basis for field usability. While some projects will need to collect more sophisticated GeoJSON objects (lines, linestrings, and polygons), we anticipate that support for the collection of these objects will happen in a later development cycle – not because the library does not support it, but because the user interface needed to specify and interact with complex geometries will consume significant development time and is not part of the three modules we are targeting as part of this development cycle. However, all of the fundamental data structures support a more sophisticated mobile GIS – which we develop with successive rounds of funding.

However, more projects will need to visualise external vector data. To increase GIS vector display performance, we hope to use Leaflet.VectorGrid or Leaflet.VectorTileLayer as a way of precomputing and rendering these vector tiles. However, we did not have enough built during elaboration to justify significant exploration of GIS capabilities in this phase. As a development priority, however, we will focus more on GIS performance than features.

## Dynamic Plugin Architecture

Because we will be building a 'progressive native web application' written in Javascript, we will rely heavily on a dynamic plugin architecture. As many aspects of the application as possible will ship as plugins. This approach provides a modular structure that avoids vendor lock-in. It also supports custom functionality required by clients without recompiling the application. We hope to be able to have modules load custom, module-specific, plugins downloaded from a central server. For example, instead of compiling Zebra bluetooth printer support into the core FAIMS application, we hope to load the necessary bluetooth-serial plugins dynamically when users request the CSIRO module. This will improve performance for modules that do not require the additional functionality and allow us to more easily customise specific user experiences. Our intended plugin architecture will support Javascript and webassembly.

We have not significantly elaborated on our plugin architecture. While there are indications that Android and iOS support dynamic loading, there is no evidence that Capacitor integrates into these specific native code features. We hope to also support Java (Android) and Swift (iOS) dynamic plugins, but that hope is not currently explored in the literature.

Where possible, we will strongly prefer Javascript plugins for maintainability, although some tension exists between Javascript and native binding-reliant elements. If we use dynamic, native-code plugins, they will have to be transpiled into objective C and Java as needed. Only in the last resort will we write new Objective C or Java. Managing these development time tradeoffs will be one of the major factors of plugin versus core functionality.

## 2.3 Application Programming Interfaces (APIs)

One of the limiting factors of FAIMS 2.6 was the lack of interoperability with external programs, requiring dedicated on-server 'exporters' to transform module data and produce files compatible with spreadsheets, ArcGIS, and Google Earth. FAIMS 3 will instead support a modern API for data exchange and other interactions.

### 2.3.1 Datastore access

A highly normalised, append-only relational database was required in FAIMS v1.0-2.6 to support profound customisation and robust versioning, but it made data export difficult and data import nearly impossible. In FAIMS 3.0, we will open our data to external services. We plan to use an unmodified CouchDB instance as our server-side datastore. CouchDB uses well documented RESTful APIs to create, read, update, and delete data. Therefore, most external interactions with our server do not involve the mobile application; users only need to know how we store record 'documents' in the database.

### 2.3.2 Application API

The FAIMS 3 application, however, should function both as a data collection app and a 'server,' handling module coordination and data synchronization within the same codebase. As a result, individual instances of the app will need to be able to communicate to each other: changes in data, modifications in vocabularies, and sometimes even acting as a server, given an instance of the app running inside a cloud virtual machine.

Thus, the FAIMS 3 app will support an API allowing module creation and access alongside data creation, reading, updating, and deleting. While access to the CouchDB datastore will be permitted, some partners may prefer to engage with a single, consistent record-level API rather than implementing record-parsing inside of their systems.

### 2.3.3 'Round trip' functionality

At present, there are too many dependencies in any 'round trip' between FAIMS and data editing or analysis software to decide on technology. Interaction with the Application API will likely be the basis of any data 'round trip' with properly configured external programs. Our fallback is provision of a tabular multi-record editable data view with interactive editing (i.e., something like a spreadsheet). Libraries that can interact with 'excel workbooks' suggest the viability of this approach. Potential integration with desktop applications and existing online services requires further exploration.

There are also indications that the Web Feature Service specification offers a way to expose geospatial vector features in an *editable* fashion to GIS software. While we have not investigated how to engage with this specification, we are optimistic that some sort of GIS round trip is theoretically supported by external vendors.

### 2.3.4 Data exporters

An exporter is a data manipulation sequence, so it is either implied in the round-trip functionality, or is well demonstrated by FAIMS 2.6 exporters.

## 2.4 DevOps, QA, CI/CD

### 2.4.1 DevOps Philosophy

Tooling and infrastructure choices have been made to minimise system administration and server requirements. As such, exploiting open-source development incentives offered by GitHub, Atlassian, and BrowserStack were fundamental to our approach. GitHub's offerings for Open Source Teams, Education, and public repositories allow us to run our CI/CD pipeline for free along with maintaining a robust set of version controlled repositories. Atlassian's Open Source Project License allows us to use a professional Jira and Confluence instance for free, hosted by Atlassian. BrowserStack for Open Source allows us to run device level end to end and integration tests on real and virtual devices. Utilising these software-as-a-service offerings allows us to demonstrate a software delivery pipeline that can be maintained long term by 0.2-0.4 FTE.

The next major constraint was to automate testing to the extent possible. One flaw with FAIMS v2.x was that updates required more than a week of manual testing. This manual workload rendered regression testing expensive and performance testing or end-to-end tests impossible. As FAIMS3 is designed to be self-service – from module generation to data export – automated end-to-end testing was a fundamental requirement.

Testing responsibilities will be divided between the teams. Our programmers will be responsible for writing unit tests in Jest. These Javascript unit tests will test low-level code functionality that does not interact with the user interface. Our external QA team will be responsible for integration and end to end testing. This higher level testing will ensure that all the components work together, that the user interface is tested, and will make sure that features as specified in user stories are exercised. CSIRO will also be responsible for double-checking our code documentation, to make sure it is informative to someone without immediate access to the developers. This approach will create a project that will ensure a level of professionalism and robustness that allows for deployment and operation at scale – and may attract external developer contributions. This design decision, however, reduces feature development due to the higher quality assurance demands placed on each feature. We believe that the tradeoff is worth it for software being developed for a five-year lifespan.

### 2.4.2 Quality assurance

Unit testing will be written in Jest and run as part of a Github Actions pipeline. Unit tests will be evaluated as a pre-commit hook, using GitHub actions on commit and pull requests. Unit tests will also be discussed as part of sprint planning and demonstrated during sprint demos. One measure of quality assurance is code coverage, how many lines of code are 'exercised' by Unit Tests. By measuring code coverage of unit testing as one of our development goals, we can hopefully make sure that we have a robust and comprehensive testing suite at all levels – suitable for five or more years of maintenance. Unit testing, however, does not include UI testing, integration testing, regression testing, or end-to-end testing. CSIRO will be responsible for developing the integration tests that will provide assurance that all components fit together and deliver our intended outcome. One design goal is to ensure that these integrated tests can also function as part of an end-to-end testing regimen so that the entire application can be exercised nightly. Unfortunately, test development is time-expensive, so we anticipate some features to require manual regression testing despite the planned end-to-end framework. Features that depend on external devices or sensors are especially likely to require manual testing. Nevertheless, if we can minimise manual testing when shipping a new version, we can release more often and with more confidence.

### 2.4.3 CI/CD Pipeline

Github Actions will provide our fundamental continuous integration / continuous delivery pipeline, compiling on commit, running Jest Unit Tests, using Capacitor, Fastlane, and Electron to build a PWA, desktop installers, and producing signed code ready for deployment to Google Play and the Apple AppStore. These binaries will then automatically upload to BrowserStack App Automate for integration tests and end-to-end testing.

We are using the generous Open Source options for both GitHub and Browserstack for this purpose, which should provide sufficient build minutes to maintain a good testing pipeline at a very low cost.

# 3 Non-elaborated functionality

## 3.1 Anticipated but untested functionality

We did not test extended Capacitor functionality because we needed to prioritise basic data structures, language features, and map-showing functionality. Documentation, however, indicates that the following required functionality exists as Capacitor plugins:

- Basic geolocation: https://capacitorjs.com/docs/apis/geolocation
- Basic file access: https://capacitorjs.com/docs/apis/filesystem
- Basic camera: https://capacitorjs.com/docs/apis/camera
- Save/play video: https://github.com/capacitor-community/media

If we choose to include Ionic as well as React in our dependencies, we would increase our dependence on external vendors and libraries they choose to import, but we would gain additional features for little development time:

- A barcode scanner (CSIRO): https://ionicframework.com/docs/native/barcode-scanner
- Raw Bluetooth serial connections for mobile printing (CSIRO): https://ionicframework.com/docs/native/bluet serial
- A document scanner (Oral history for PICF forms): https://ionicframework.com/docs/native/document-scanner
- Audio capture (Oral history): https://ionicframework.com/docs/native/media-capture

External GPS support (a common requirement) would require us to fork, adapt, and support a Cordova plugin and Javascript library:

- https://github.com/heigeo/cordova-plugin-bluetooth-geolocation
- https://github.com/infusion/GPS.js/

Basic GIS analytics (Lake Mungo for finding grid squares) requires:

- https://turfjs.org/docs/#intersect

## 3.2 Functionality not yet elaborated

FAIMS 3 has many planned features that do not exist as libraries to include within our planned Javascript framework, or that depend so strongly on infrastructure and architecture decisions that anticipating a solution now would be premature. Functionalities that we anticipate but have not yet elaborated include:

- Offline Map loading and serving. One option is utilising something like the Shapefile Package during module generation to convert vectors into GeoJSON, plus an offline tile server. Offline maps, GIS, and map generation is a part of FAIMS 3 that could consume extensive resources and it must be carefully managed. Both CSIRO and Lake Mungo require the display and management of vector and raster layers, but we have not yet explored trade-offs between options. Leaflet has demonstrated that it can render offline maps and points, so our fallback will be to run a 'normal' map tile server and then allow caching of maps on the device.
- UI elements (especially 'dynamic' UI). We have not yet explored interface elements like 'tabs' and 'tab groups', including the ability to dynamically alter input fields based on user interactions, as used in FAIMS v2.x. Dynamic manipulation of HTML is an extremely mature technology.
- GUI module generator – depending on complexity, we will need to decide between schematic representation (lists of elements) versus a more WYSIWYG editor. However, developing a standard web application which does not require offline support which can generate a module specification through some means is not of concern.
- External data importers are not currently planned during this development cycle, but may be implied by the round-trip functionality or, at minimum, developed as a later plug-in (accommodated by the architecture discussed above).
- On-load Internationalisation is well demonstrated in FAIMS 2.6.

OSGeo integration. Use of GeoCouch, GeoTools, and GeoServer have implications for map-serving and roundtrip capabilities.

# 4 Team Composition and Development Plans

FAIMS 3 will be developed by three groups: FAIMS leadership, AAO, and CSIRO.

- FAIMS leadership will be responsible for planning and execution – development planning before each development period, assessing development via a demo after each period, providing subject matter expertise, and writing user stories and appropriate acceptance criteria for the stories. Leadership is also responsible for DevOps and ultimately ensuring that all the systems work.
- AAO will be responsible for primary development – implementing agreed user stories each development period, performing demos on a regular interval (every two weeks), writing unit tests, and documenting their code so that external developers can contribute.
- CSIRO will be responsible for QA – writing integration and end-to-end tests against AAO's user stories and documentation. By shipping code across the country, requiring that a second team understand, execute, and test the code, we hope to avoid testing situations where the testers lean across the office corridor and ask the developers for help rather than documenting usability or documentation deficiencies.

We will (mostly) follow an agile development methodology informed by the Rational Unified Process. In short, each 'sprint' will be preceded by a development planning meeting where user stories are allocated and a general work consensus is achieved, ensuring that the plans are reasonable, with time allocated for testing, documentation, and bug fixes. No formal point/time allocation per story will be part of this process, only informal estimates to better calibrate expectations to development pace. After each two-week work-block, each story worked on will be debriefed: working tests will be demonstrated and problems will be discussed as a way of improving future planning. Where features are interesting to the larger community, FAIMS Leadership will turn the demonstration/retrospective into a regular blog post.

Everyone involved is very aware that agile development means that we spend money for a specified period of time, rather than agree on a formal waterfall Software Development Life Cycle. Development thus intends to achieve the design objectives stated above, though some descoping will likely be required, as we are prioritising quality assurance over feature development.

Each external milestone will be preceded by User Acceptance Tests. These tests will be written with input from FAIMS leadership, AAO, and CSIRO so that external assessors can work through the features that best demonstrate development progress. Only when everyone is satisfied that the tests have been passed will software be released to external users.

# 5 Future Decisions

## 5.1 Dependency hell is a place

The fundamental thing we need to decide is how much external code to use in the FAIMS3 'core'. Inevitable external code dependencies exist in Bluetooth, data transport, and GIS libraries, but keeping the core free of excessive Javascript packages will increase maintainability at the cost of longer development time.

We could, however, decide to commit to React + Ionic (Community or Enterprise) + Capacitor + JSON Forms + 'all of the plugins'. This route gets us many more features at the cost of being beholden to many different communities for maintaining their pieces of code.

We will almost certainly thread a needle between these extremes – but wish to acknowledge the tradeoff explicitly. We need to deliver the required features, but avoid being trapped by an unsupported package that exposes us to a security vulnerability or major component failure.

## 5.2 GIS Features

We need to determine what subset of GIS functionality is actually used in the field by our users (as opposed to what users claim they need), and what external connections we need to support desktop GIS applications. We developed many expensive GIS features in FAIMS 1.0-2.6 at the urging of researchers that were never used in the field, an expensive error we must not repeat.

## 5.3 The Round Trip Problem

We need to figure out what users require from 'data round trips' and how much can be delivered in-application versus exporting bundles of CSVs. There are some fundamental issues with data round-tripping: denormalisation, preserving internal identifier row-associations, preserving vocabulary keys, and preserving GIS-appropriate associations. We also need to be somewhat program-agnostic, so that users can bring the programs they are used to working with to the field camp for evening analysis. There is also a risk of building very technical bridges to specific programs that will end up being ignored by users. Determining appropriate compromises to allow for bulk data review and editing – tabular, multimedia, and geospatial – will be a significant challenge determined by our data structure and development choices.

## 5.4 External Partner Support

FAIMS 3.0 must interact with some of our external partners' infrastructure: OpenContext, tDAR, and Cloudstor. At present we plan to deploy well documented APIs to access the FAIMS 3.0 application and data in the CouchDB instance.  We also plan to add API function calls as requested by our partners to support a more interactive export style.  One example could be that a user from one of these external services enters a FAIMS 3.0 application URI and authenticates appropriately – and then the service can prepare their data for export and ingest into their own system. Planning where the bulk of data preparation will occur, however, requires a better sense of our final architecture and conversations with our partners. We plan to begin an external service elaboration when FAIMS 3.0 enters its beta version User Acceptance Tests. At that stage we will explore scenarios with each partner to accommodate user needs.

# Bibliography

Ballsun-Stanton, Brian, Shawn A Ross, Adela Sobotkova, and Penny Crook. 2018. "FAIMS Mobile: Flexible, open-source software for field research." *SoftwareX* 7 (January): 47–52.

Bureau of Reclamation. 2017. *DataApp: A Mobile App Framework for Field Data Capture.* https://www.innocentive.com/ar/challengeWorkspace/challengeDetails/655716. Accessed: 2018-3-27.

# 6 Technical Advisory Group Comments and Responses

From Kate Robertson, email:

> I've had a look at the document, no comments/changes etc from me, looks good- thorough and well explained.

From Richard Adams, email:

> Just a few specific points -
>
> 1) We've found react.js to be excellent in our project and we are committing to refactoring all our UI code to use react. It has such a vast usage that it's extremely unlikely to get abandoned over the next 5 years or so. The problems it solves such as easy reuse of components enabling a more standard UI has made a vast difference to the appearance and behaviour of our application, and even if initially slower to get started with, bears fruit over time as new pages can often be composed of existing components
>
> 2) The API - not sure if this API is an 'internal' API for instances of the app to communicate and exchange data with each other/ or the local server, or a public API for 3rd party software to access the data. If the latter, have you considered having an intermediate data layer so that the API isn't tied to the underlying data structures in CouchDB, and the API and the internal data structures can evolve independently of each other? This would help with maintaining stability of the API for 3rd-party developers
>
> 3) Testing and QA
>
> Always a thorny issue. For RSpace, we initially wrote manual test scripts for all features, then employed someone to automate them as much as possible using Selenium. This is an enormous effort to maintain; the tests can be fragile and with over 300 of them the false failure rate is difficult to keep to acceptable levels. For a new module we are instead doing more exploratory testing as means to identify defects; performing manual, unscripted testing for fixed periods of time. If the defect rate increases, we do more testing; as the defect rate falls we do less testing. Expected behaviour is defined in the use-cases and requirement specs; we use decision tables and state matrices to formalise key behaviours and states. We found this cuts down a lot of duplication of writing out requirements, then writing a largely similar testing document.
>
> The aim is to do 'just enough testing but no more' ie- how little acceptance testing can we do, before we get an unacceptable defect rate? So far, results are encouraging. Also, for software designed to be used by humans, manual testing by humans is essential, and actually beneficial due to insights into user-experience that automated testing would ignore.

Having said that, there is some scope perhaps for a small number of automated integration 'smoke tests' that can detect egregious defects on a nightly build. We recently evaluated Cypress.io and liked it as a possible successor to Selenium for test automation; I understand they are developing a component-testing framework as well.

Code coverage is OK for catching gaping holes in test coverage - for example finding whole files or components; but a coverage tool has to be able to indicate branches and statements covered as well as just lines, in order to make sure that infrequently-called blocks are also tested ( for example exception/failure handling code).

Are you planning to do code-reviews as part of your Git workflow, e.g before merging a feature branch to mainline? We have found these useful; people really improve their code if they know their colleagues are going to comment on it; and it also implicitly spreads knowledge around the team and helps conventions to become established. Obviously this depends on the size of the team, but even in our tiny team of 2 front-end and 2 backend devs, code quality and consistency has markedly improved since we started requiring manual code review before merge.

You mention that performance testing will be possible with this new process but it's not elaborated further - are some performance goals going to be established? I expect a mobile app in the field, efficient power usage is essential to maximise battery life, is that a design goal at all?

4) Devops

This all sounds great. Automation of the build and deployment as much as possible is definitely the way to go. As opposed to manual testing, there are almost no benefits to be obtained from manual builds.

5) Project schedule

Re the agile approach - are these going to be internal sprints, doing a number of iterations before public release, or have you considered doing very early alpha releases and inviting keen users to try out the software, even before reaching Minimum Viable Product stage? We've been trying this approach for a new product. It takes some time to coordinate the volunteers, manage expectations and organise the feedback but we've found several benefits - important features that we had missed out on were detected; user experience on a variety of devices; public visibility that the project is making progress.

6) General questions

How does data collected by FAIM[S] get linked to non-field related data for a project? For example, for an archaeological project, there will be the field data and also perhaps laboratory data examining the artefacts - would all this be put in FAIM[S] database or would a research team use other data management software too? My interest here is if an ELN could be used to tie everything together.

7) This sounds like a fascinating project. I'd be particularly keen on helping out or testing any external APIs to work with FAIM[S]-acquired data, and of course happy to follow up with more information on any of the testing/ PM ideas I mentioned above.

Response to Richard Adams

1. Thanks for the advice regarding React. We plan to have some sort of consistent UI framework. Steve has explored a version of the elaboration prototype without react and has reported some significant size/complexity reductions. However this will be dictated by what specific components React offers us. The dependencies required by react are a non-trivial cost.

2. Right now the plan for the API is to be external-facing, as we are trying to avoid needing to code and maintain a "server" on top of everything that FAIMS 3 will already be. Some light data abstraction layers, or at least data-format-consistency is desirable and will likely be requested by our external partners. How we're going to achieve that data consistency is not yet determined.

3. We unfortunately have had the opposite problem with testing and QA. My goal for this testing regimen is to avoid after-hours, emergency tech support over poor internet connections to projects in the field. Again. The risks of data-loss and the costs of patching are much higher for an offline system like ours. Beyond that, human costs while testing are one of the reasons why FAIMS 2 aged so poorly, as we were not able to sustain maintenance development with the week-plus regression tests needed. Conversely, we recognise the costs inherent to the proposed QA approach. My goal is to automate as much QA as we can afford – to minimise data-loss and emergency tech support to offline (or marginally online) projects in remote areas. This challenge is somewhat different from supporting users who have normal access to their servers and infrastructure.
   (a) Yes, we also desire code-reviews. The specific pragmatics of the situation have yet to resolve.
   (b) The performance goals are mostly in relation to data capacity and access. FAIMS 2, due to the append-only datastore, had issues with high numbers of complex records. Automated performance testing (which we hope to implement again) allowed us to estimate when performance would drop off (i.e., at how many records) and plan projects appropriately. We hope the use of a noSQL database and other changes will mitigate these performance issues, but want to make sure, since we can't rely on online storage or processing during data collection and are constrained by device capability. In our requirements for FAIMS 3, we target 'ten times the number of records than in FAIMS

2' – tens to hundreds of thousands of rows.
For mobile battery, the primary consumer of the battery is the screen (some optimisation may be possible here, but is probably outside of our budget). With the partial exception of tracklogs (i.e., constant GPS use, especially under canopy) on some devices, we had very few problems with battery life in FAIMS 2 and do not expect any to arise (user testing should reveal any such problems early on).

4. That is also our understanding (little or no benefit from manual builds), and why I will personally be responsible for automating DevOps as much as possible.

5. We currently anticipate internal sprints. Our project governance arrangement includes 'ad hoc user panels' who will advise about features and also complete UATs. We think it will be possible to involve them in pre-alpha testing of discrete components of the system as they become minimally operational, offering an opportunity for early feedback. We did something similar for FAIMS 2 and it was very valuable. The user panels will certainly be involved in alpha testing. We will be open about progress and feedback, with a public road map (as per Rory Macneil's recommendation, see below).

6. Data interoperability is important to us, see discussion around 'external round-trip' as one of our design goals for this version. We would love for an ELN to be one of our partners and to explore interoperability. In terms of dividing data capture and management between pieces of software, we are focusing heavily on data capture in field contexts, and assume that researchers would use more appropriate tools for other data capture scenarios. We then want to provide a capacity for data export in a way that will make the data as easy to reconcile (either by ingest into the other system, or into a third system for combining data). To address your example, artefact analysis is something that we've considered a 'core' activity that we'd do in FAIMS (since some of that analysis happens in the field / offline). Other analysis that usually happens in a lab (say, palynological analysis, radiometric dating, stable isotope analysis, etc.) would be better covered by an ELN built more for that purpose. If the environmental samples for such analysis were recorded using FAIMS, we could connect the data by (for example) assigning a persistent identifier (PID) in the field (e.g., an IGSN), printing a bar or QR code with the PID to include with the sample, scanning in that code in the lab, completing the analysis, and then reuniting the field and lab data based on the PID. Or, if it were easier, we could export the field data for ingest into the ELN as sample metadata. We're focusing on doing one thing - offline data capture - as well as possible, and then federating with other systems to do other things. Relating back to your API question, we are aiming for an API that

3rd-party apps can interact with, and would like to pursue interoperability with an ELN to allow integration of field and lab data. Ideally this interoperability would go both ways - it can be useful to have lab results when out in the field on subsequent fieldwork. As Brian mentioned, we're trying something analogous with Cloudstor (OwnCloud) interoperability (to allow online viewing / editing of data in OpenRefine or a GIS).

7. Thanks! When we have APIs for user acceptance testing, we will make sure to let you know.

From Jeff Good

I read through the document and found it to be very well explained and thorough. The choices seem well justified to me (though I can't speak to direct knowledge of many of the technologies).

From Nathan Reid

I have read through and am happy with the document. The flow makes sense and I understand why certain options were chosen.

From Jonathan Smillie:

1. 'The code and platform should last for at least five years, assuming regular maintenance'. [Is this an] expectation, or requirement?
2. Is 'server' here a central server per-end-user deployment? Or a single central server hosted by the FAIMS platform project itself? (Or are app instances acting as distributed peer-to-peer servers?)
3. [Is] Export via app, or direct from server-side CouchDB instance?

Response to Jonathan Smillie:

1. Design expectation. A 5-year build time is based on our experience with FAIMS 2 and the pace of change in mobile technology. We have geared our business plan towards this expectation.
2. The 'server' represents a client's main datastore. Our hope is that each instance of the app can act as a 'server' (since that will mean that we only need to code one

app). While we will be offering hosted (online) 'servers' for clients as part of our sustainability plan, being able to deploy offline to local hardware is essential for off-the-grid research teams.

3. We have not yet specified export mechanism. The answer, we suspect, is 'all of the above' – We are likely to use one or both of the approaches you suggest, plus our plugin architecture implies that data-manipulation javascript can be loaded as part of the client. Partner organisations like AARNet and the data repositories will also be working with us on export capabilities.

From Rory Macneil, email and conversation (excerpted / paraphrased):

1. Ensure that development is market-led / customer-led rather than technically led by involving users or clients at every stage of testing, getting feedback early and often (and responding to it).

2. UI/UX is everything (for uptake and ultimate success). Invest in it appropriately. UI/UX can serve as a bridge between the development team and the commercial or client-facing side of the operation. Do detailed documentation as you develop each feature, involving the dev team. Publicise development early, providing a roadmap. Make short videos about new features. Alongside involving users in testing, these activities can build a community of interested people who are potential customers.

3. Investigate tools for customer engagement. we use Intercom (statistics on users and usage), Simpo (on-boarding tool for non-technical users), HelpDocs (documentation), noting that these tools integrate with one another.

4. During development, RSpace (specifically referring to the Inventory Hub product) provides:
   - Two-minute videos covering key features in each release.
   - An easy-to-understand Roadmap (on Trello NOT github because our users are not developers).
   - Full user documentation of all features as they are developed.
   - Professionally designed and delivered user testing sessions.
   - Sandbox instance where anyone can try out the latest version.

5. Design your permissions system early; SSO integration is painful (but necessary for enterprises).

Response to Rory Macneil

Thank you! We have consciously attempted to ensure our development is market/customer-driven, but this reminder is timely. We have also used some of these approaches in the past, but not all of them, especially as part of a ‘package’ like you suggest. We will work to incorporate these suggestions, although some will require additional resourcing (which we are currently seeking).